Unit 4

Transaction Processing, Concurrency Control and Recovery

Why you should Learn Transaction Processing in Database ?

What is Transaction ?

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	**Name**	**Balance**
501	Ram	150
502	Shyam	100

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Transaction:
Transfer Rs. 100 from Ram account to Shyam account.

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Transaction:
Transfer Rs. 100 Ram account to Shyam account.

Question:
What are the SQL statement to be executed to perform the above transaction ?

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Transaction: Transfer Rs. 100 Ram account to Shyam account.

Subtract Rs. 100/- from Ram account

```
update ACCOUNTS
set Balance =Balance-100
where AccountNumber=501;
```

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Transaction: Transfer Rs. 100 Ram account to Shyam account.

Subtract Rs. 100/- from Ram account

```
update ACCOUNTS
set Balance =Balance-100
where AccountNumber=501;
```

Add Rs. 100/- to Shyam account

```
update ACCOUNTS
set Balance =Balance+100
where AccountNumber=502;
```

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Transaction: Transfer Rs. 100 Ram account to Shyam account.

Subtract Rs. 100/- from Ram account

```
update ACCOUNTS
set Balance =Balance-100
where AccountNumber=501;
```

Add Rs. 100/- to Shyam account

```
update ACCOUNTS
set Balance =Balance+100
where AccountNumber=502;
```

ACCOUNTS

Account Number	Name	Balance
501	Ram	50
502	Shyam	200

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Transaction: Transfer Rs. 100 Ram account to Shyam account.

Subtract Rs. 100/- from Ram account

```
Update ACCOUNTS
Set Balance =Balance-100
Where AccountNumber=501;
```

FIRST

Add Rs. 100/- to Shyam account

```
Update ACCOUNTS
Set Balance =Balance+100
Where AccountNumber=502;
```

SECOND

Question:
What will be the status of **ACCOUNTS** table say if **FIRST** Update SQL statement has been executed but **SECOND** Update SQL statement **has not been executed** because of electricity failure on the computer system.

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Transaction: Transfer Rs. 100 Ram account to Shyam account.

Subtract Rs. 100/- from Ram account

```
Update ACCOUNTS
Set Balance =Balance-100
Where AccountNumber=501;
```

FIRST

Add Rs. 100/- to Shyam account

```
Update ACCOUNTS
Set Balance =Balance+100
Where AccountNumber=502;
```

SECOND

Question:
What will be the status of **ACCOUNTS** table say if **SECOND** Update SQL statement has been executed but **FIRST** Update SQL statement **has not been executed** because of electricity failure on the computer system.

Transactions : Introduction

- Example: Bank database application
- Consider Ram and Shyam has an account in SBI bank at Basvangudi branch.

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Transaction: Transfer Rs. 100 Ram account to Shyam account.

Subtract Rs. 100/- from Ram account

```
Update ACCOUNTS
Set Balance =Balance-100
Where AccountNumber=501;
```

FIRST

Add Rs. 100/- to Shyam account

```
Update ACCOUNTS
Set Balance =Balance+100
Where AccountNumber=502;
```

SECOND

SOLUTION:

Both **FIRST** and **SECOND** Update SQL statements should be executed successfully for transaction to complete

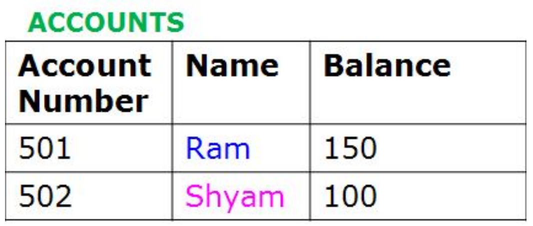
In situations when any one of the UPDATE SQL statements i.e., **FIRST or SECOND** has been executed then transaction should be **aborted** i.e., in ACCOUNTS table Balance column should not be changed

Demonstration

Placing Transaction in between START TRANSACTION and COMMIT.

Demonstration

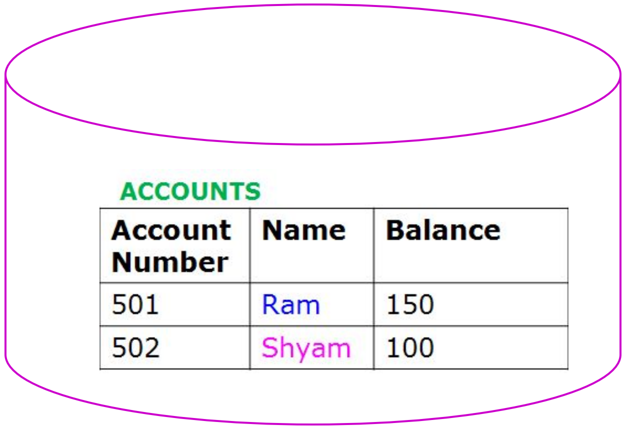
```
DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts ( account_id INT PRIMARY KEY,  owner
VARCHAR(30),  balance FLOAT) ;
INSERT INTO accounts VALUES (501, 'Ram', 150.0);
INSERT INTO accounts VALUES (502, 'Shyam', 100.0);

select * from accounts;
```



ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Demonstration



ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

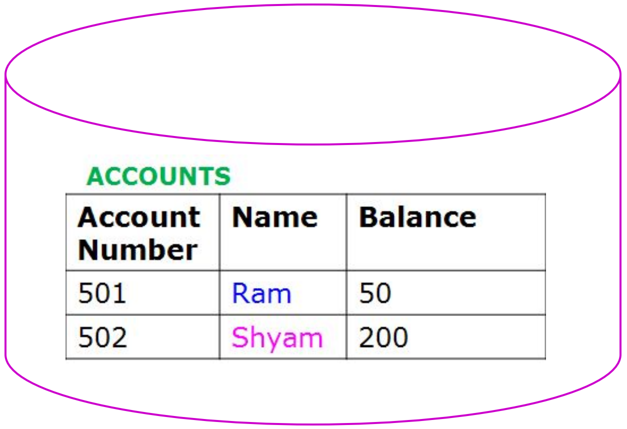
User 1

```
use bank;
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE Name = 'Ram';
UPDATE accounts SET balance = balance + 100 WHERE Name = 'Shyam';
COMMIT;
select * from accounts;
```

Demonstration

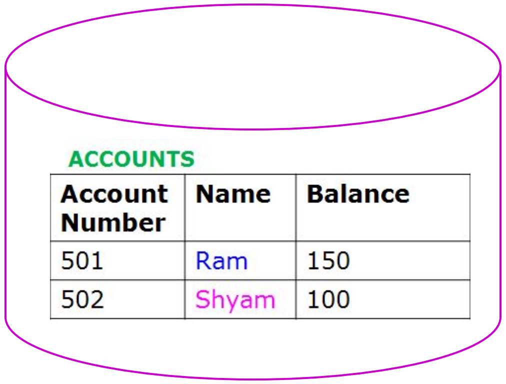
User 1

```
use bank;
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE Name = 'Ram';
UPDATE accounts SET balance = balance + 100 WHERE Name = 'Shyam';
COMMIT;
select * from accounts;
```


ACCOUNTS

Account Number	Name	Balance
501	Ram	50
502	Shyam	200

Demonstration



ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Bank database

User 1

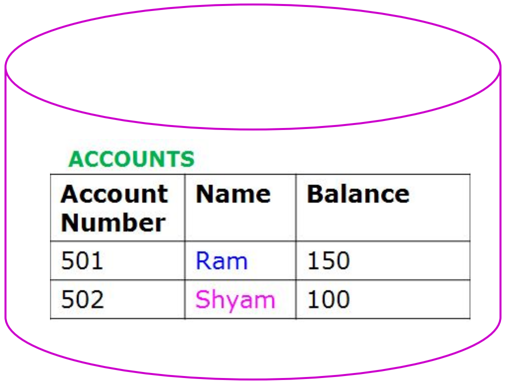
```
use bank;
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE owner = 'Ram';
UPDATE accounts SET balance = balance + 100 WHERE owner =
'Shyam';
COMMIT;
```

User 2

```
Use bank;
select * from accounts;
```

Question:
What will be the balance of Ram and Shyam when User 2 executes Select statement on accounts table

Demonstration



ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Bank database

User 1

```
use bank;
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE owner = 'Ram';
UPDATE accounts SET balance = balance + 100 WHERE owner = 'Shyam';
COMMIT;
```

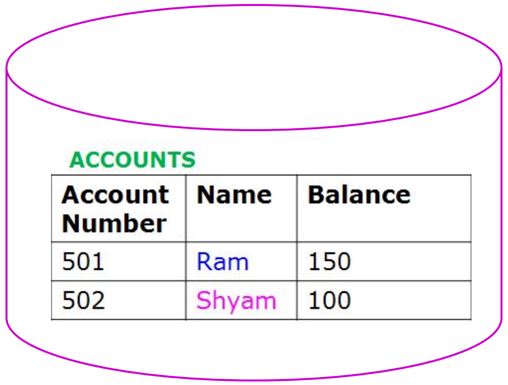
User 2

```
Use bank;
select * from accounts;
```

ACCOUNTS

Account Number	Name	Balance
501	Ram	50
502	Shyam	200

Demonstration



ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Bank database

```
use bank;
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE owner = 'Ram';
UPDATE accounts SET balance = balance + 100 WHERE owner = 'Shyam';
~~COMMIT;~~
```

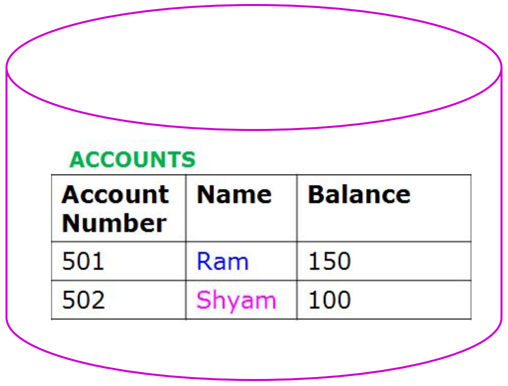
User 1

```
Use bank;
select * from accounts;
```

User 2

Question:
What will be the balance of Ram and Shyam when User2 executes Select statement on accounts table after User1 has executed the set statements mentioned above for him

Demonstration



ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Bank database

User1

```
use bank;
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE owner = 'Ram';
UPDATE accounts SET balance = balance + 100 WHERE owner = 'Shyam';
~~COMMIT;~~
```

User2

```
Use bank;
select * from accounts;
```

ACCOUNTS

Account Number	Name	Balance
501	Ram	150
502	Shyam	100

Changes done by User1
Will not be seen by User2
because User1 has not completed
the transaction i.e
COMMIT statement
has not been executed by User1

Transactions

Example:

- **Transaction:** Transfer Rs. 100 from Ram account to Shyam account.

```
UPDATE accounts SET balance = balance - 100 WHERE owner = 'Ram';
UPDATE accounts SET balance = balance + 100 WHERE owner = 'Shyam';
```

Above two Update statements must run, or neither must run. You cannot have the money being transferred out of one person's account, and then 'disappearing' if for some reason the second query fails. Both these queries form one *transaction*.

A transaction is simply a number of individual queries that are grouped together in between **START TRANSACTION** and **COMMIT** statement.

```
START TRANSACTION
UPDATE accounts SET balance = balance - 100 WHERE owner = 'Ram';
UPDATE accounts SET balance = balance + 100 WHERE owner = 'Shyam';
COMMIT
```

Transactions : Basic Definition

- A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.
- In the real world, a TXN either happened completely or not at all
- Examples:
  - Transfer money between accounts
  - Purchase a group of products
  - Register for a class (either waitlist or allocated)

```
START TRANSACTION
	UPDATE Product SET Price = Price – 1.99 WHERE pname = 'Gizmo'
COMMIT
```

Transactions in SQL

- In "ad-hoc" SQL:
  - Default: each statement = one transaction

- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 100 WHERE Name = 'Ram'
        UPDATE Bank SET amount = amount + 100 WHERE Name = 'Shyam'
COMMIT
```

Model of Transaction for 15CS4DCDBM

Note: For 15CS4DCDBM, we assume that the DBMS *only* sees <u>reads and writes to data</u>

- User may do much more
- In real systems, databases do have more info...

Model of Transaction: Read-Write Operations

- read_item(X): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
- write_item(X): Writes the value of program variable X into the database item named X.

Model of Transaction: Read-Write Operations

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.
- read_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.

Model of Transaction: Read Operations

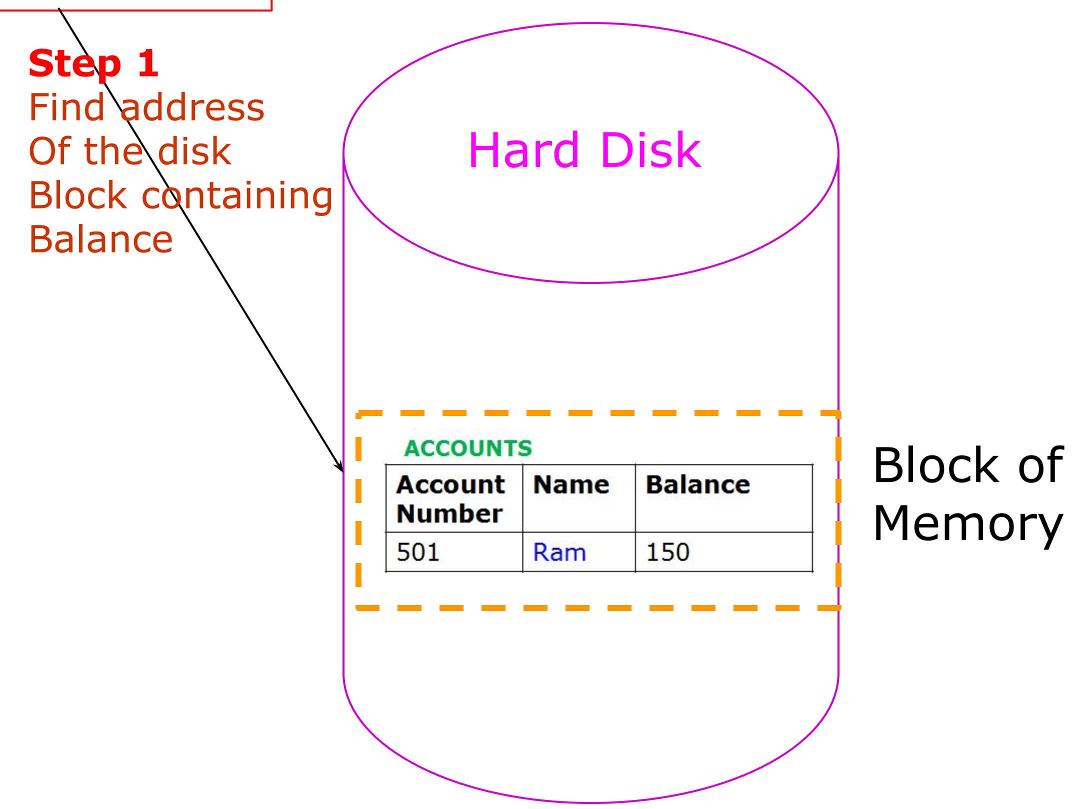
Example of transaction Read Operation: read(balance)

```
select balance from accounts;
```

Model of Transaction: Read Operations

Example of transaction Read Operation: read(balance)

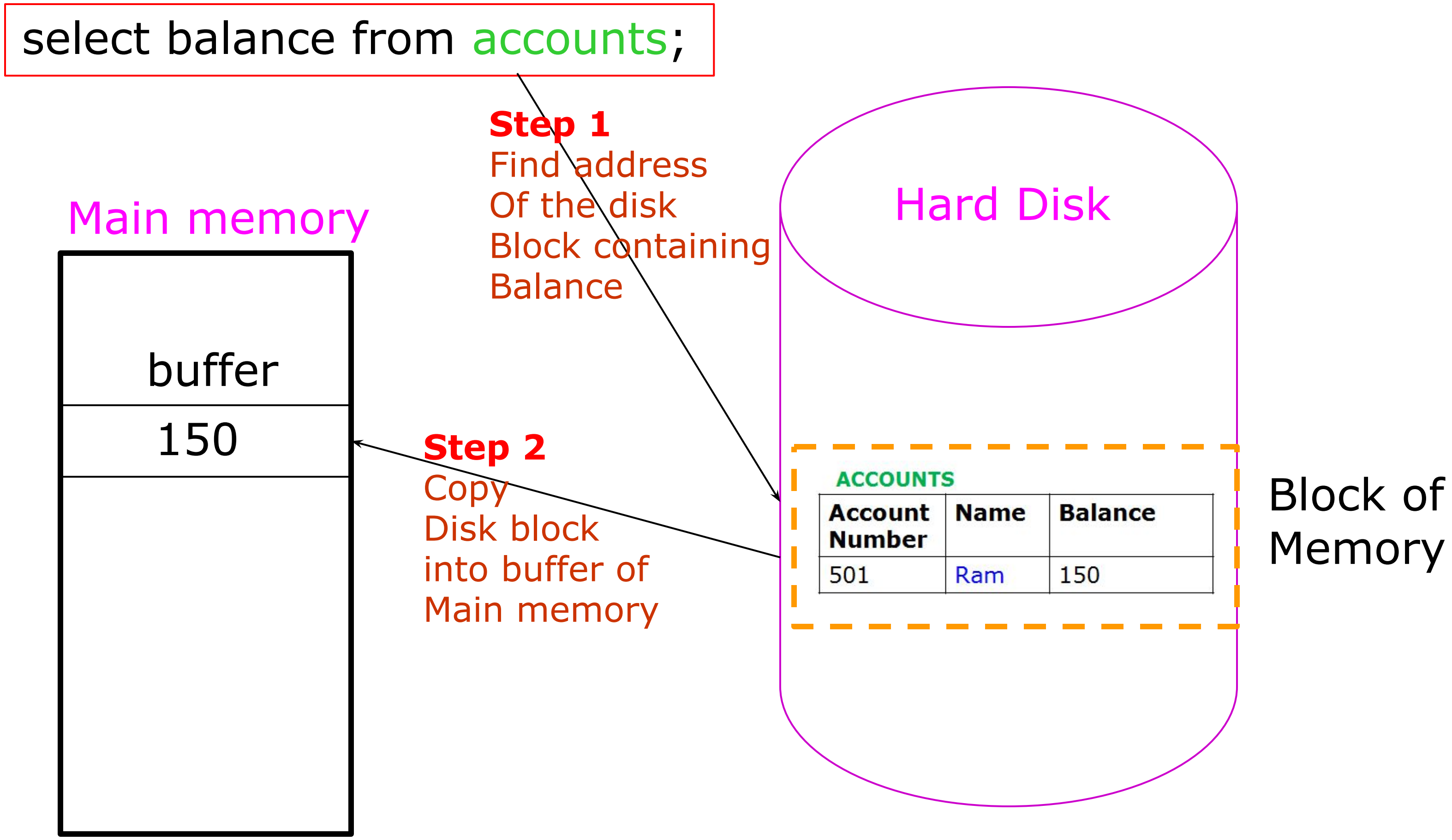
```
select balance from accounts;
```



Account Number	Name	Balance
501	Ram	150

Model of Transaction: Read Operations

Example of transaction Read Operation: read(balance)



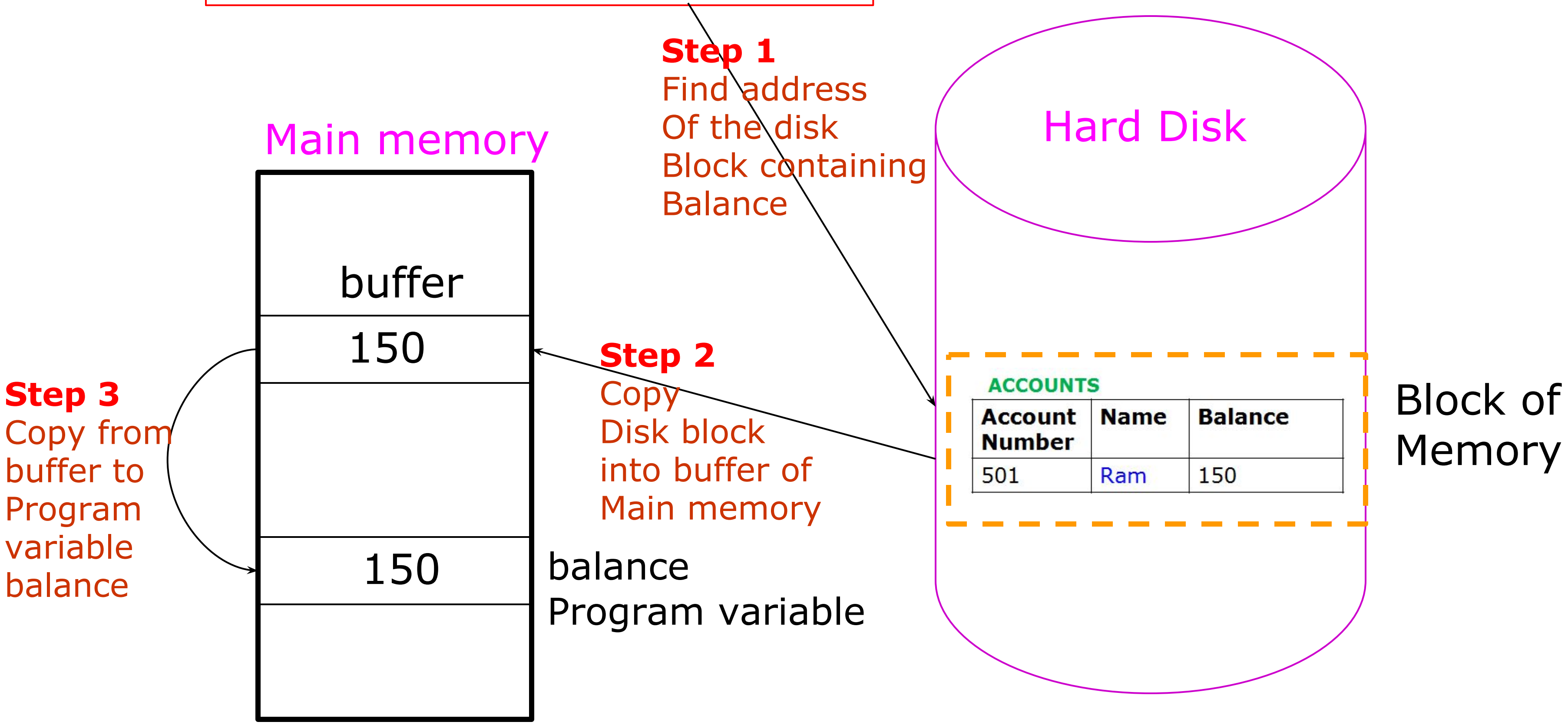
ACCOUNTS

Account Number	Name	Balance
501	Ram	150

Model of Transaction: Read Operations

Example of transaction Read Operation: read(balance)

select balance from accounts;



Account Number	Name	Balance
501	Ram	150

Model of Transaction: Read-Write Operations

READ AND WRITE OPERATIONS (contd.):

- **write_item(X)** command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Model of Transaction: Read-Write Operations

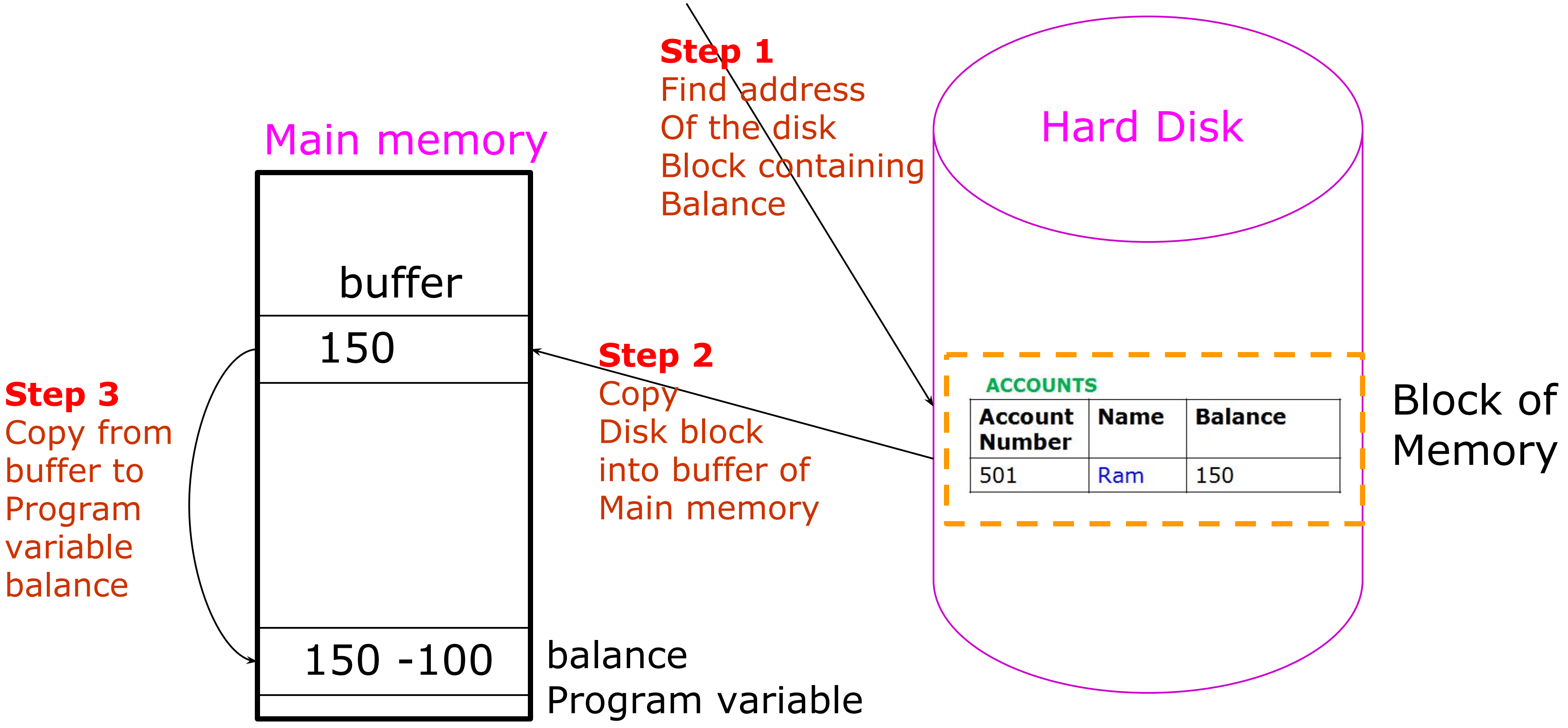
Example of transaction write Operation: write(balance)

```
Update accounts  set balance=balance-100 where Name='Ram';
```

Model of Transaction: Read-Write Operations

Example of transaction write Operation: write(balance)

Update accounts set balance=balance-100 where Name='Ram';



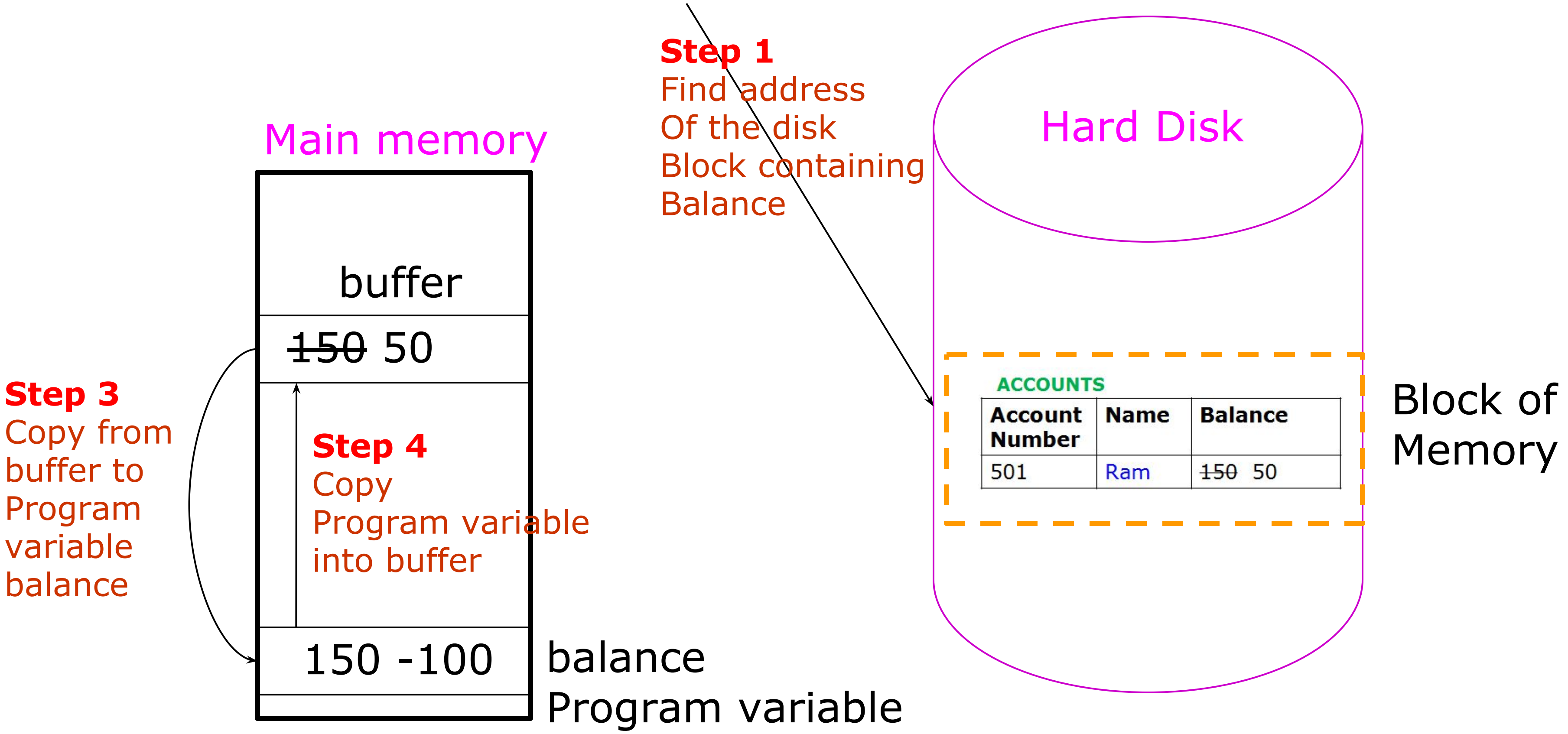
ACCOUNTS

Account Number	Name	Balance
501	Ram	150

Model of Transaction: Read-Write Operations

Example of transaction write Operation: write(balance)

Update accounts set balance=balance-100 where Name='Ram';

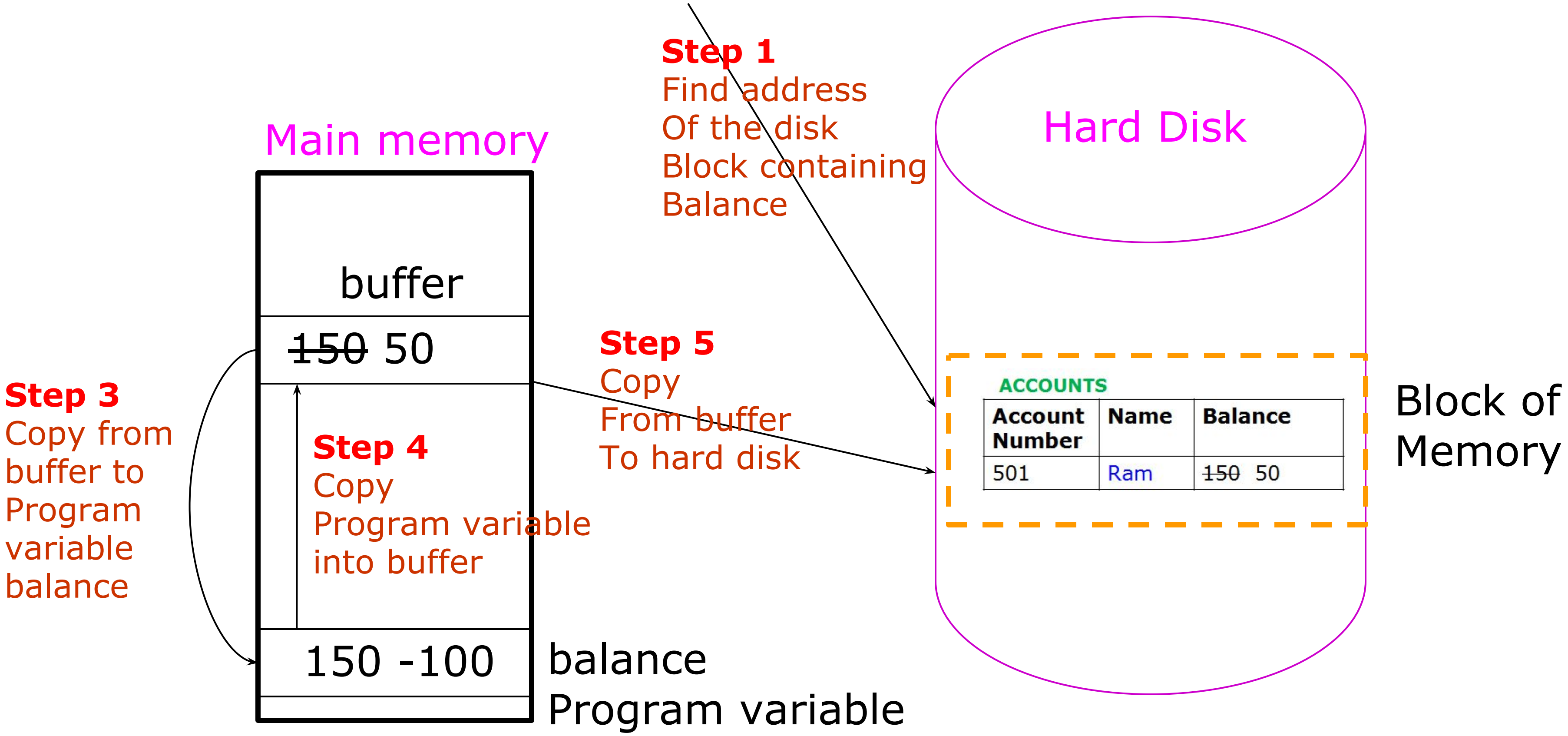


Account Number	Name	Balance
501	Ram	~~150~~ 50

Model of Transaction: Read-Write Operations

Example of transaction write Operation: write(balance)

Update accounts set balance=balance-100 where Name='Ram';



ACCOUNTS

Account Number	Name	Balance
501	Ram	~~150~~ 50

Problems with Concurrent Execution

- What is Concurrent Execution ?

When Multiple users trying to access same database record in an uncontrolled manner.

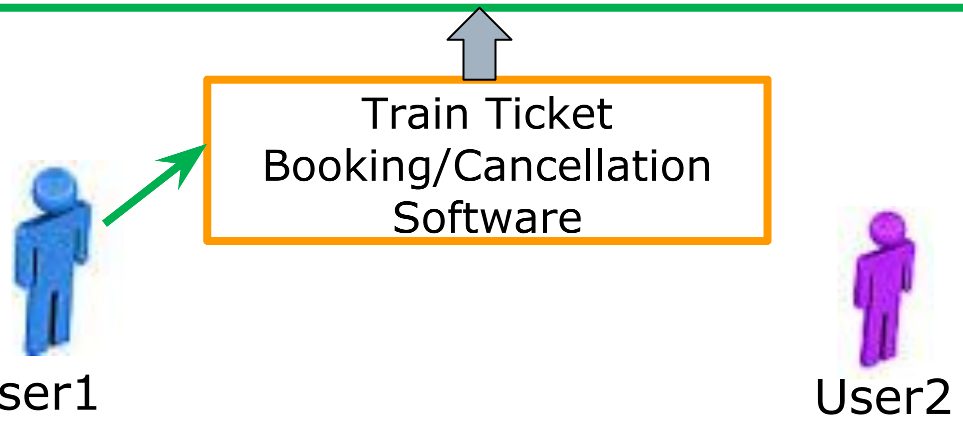
- Problems with Concurrent execution
  1. Lost Update Problem
  2. Temporary Update (or Dirty Read) Problem
  3. Incorrect Summary Problem
  4. Unrepeatable Read

To Understand problems with concurrent executions

Consider an example of Train Reservation System

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

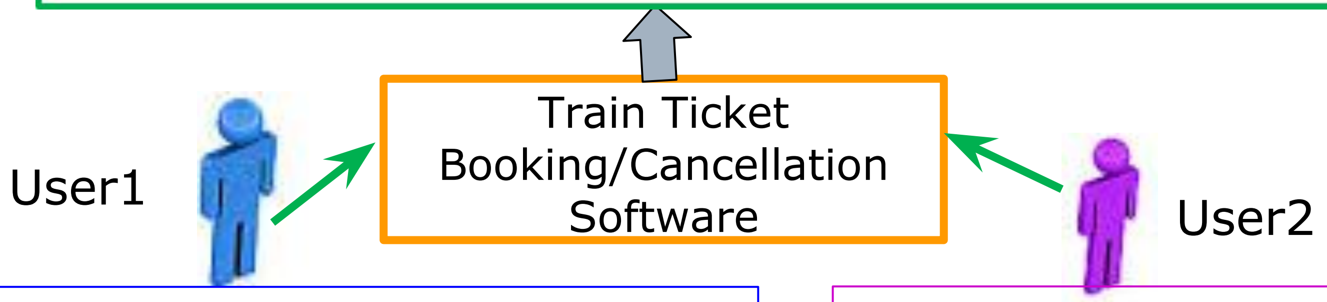


To Understand problems with concurrent executions

Consider an example of Train Reservation System

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

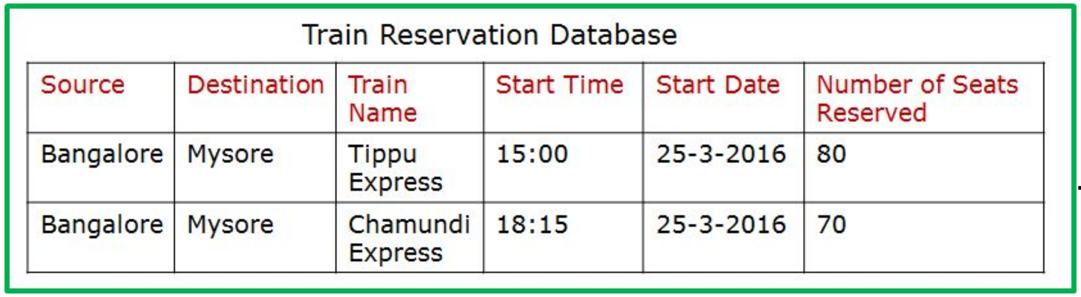


User1
1. Wants to **cancel** 5 seats on **Tippu** express
2. Wants to **reserve** 5 seats on **Chamundi** express

User2
1. Wants to **reserve** 4 seats on **Tippu** express

To Understand problems with concurrent executions

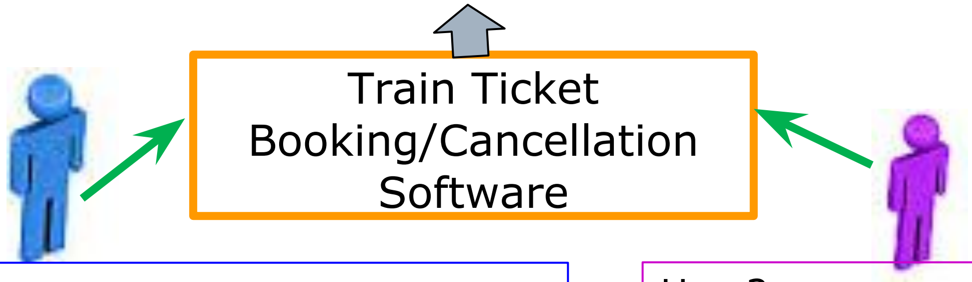
Consider an example of Train Reservation System



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

reservation table



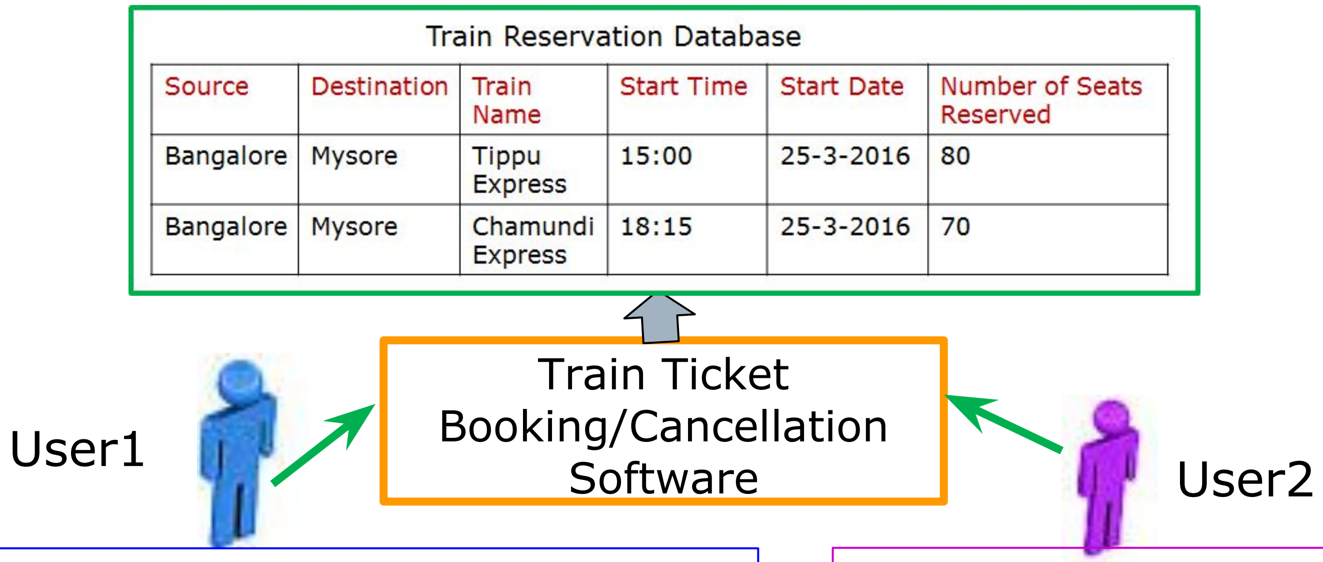
User1
1.Wants to **cancel** 5 seats on **Tippu** express
2.Wants to **reserve** 5 seats on **Chamundi** express

```
Update reservation set seats=seats-5
Where TrainName='TippuExpress';
Update reservation set seats=seats+5
Where TrainName='ChamundiExpress';
```

User2
1.Wants to **reserve** 4 seats on **Tippu** Express

```
Update reservation set seats=seats+4
Where TrainName='TippuExpress';
```

To Understand problems with concurrent executions



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

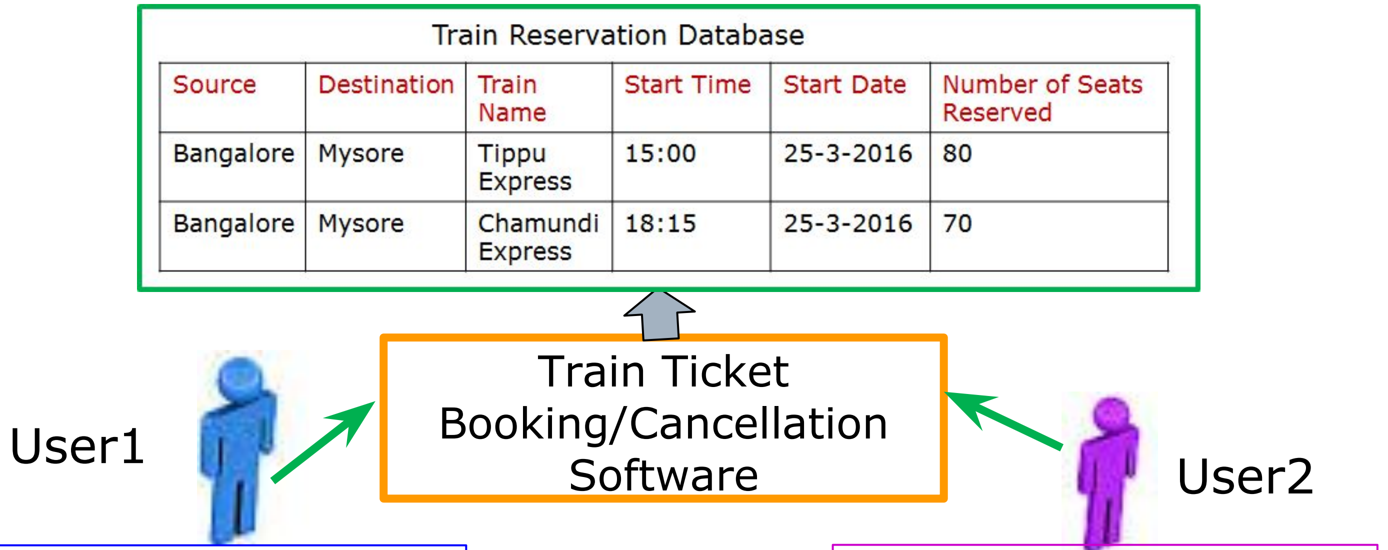
User1
1. Wants to **cancel** 5 seats on **Tippu** express
2. Wants to **reserve** 5 seats on **Chamundi** express

```
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)
```

User2
1. Wants to **reserve** 4 seats on **Tippu** Express

```
Read(TippuSeats)
TippuSeats=TippuSeats+4
Write(TippuSeats)
```

To Understand problems with concurrent executions



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

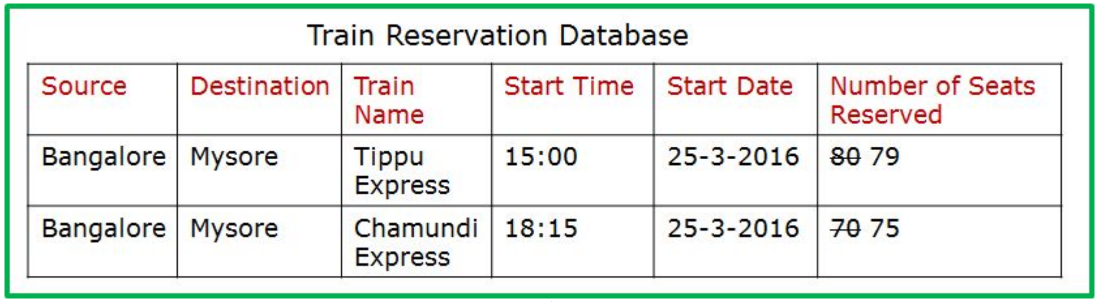
Transaction1
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)

Transaction2
Read(TippuSeats)
TippuSeats=TippuSeats+4
Write(TippuSeats)

Question

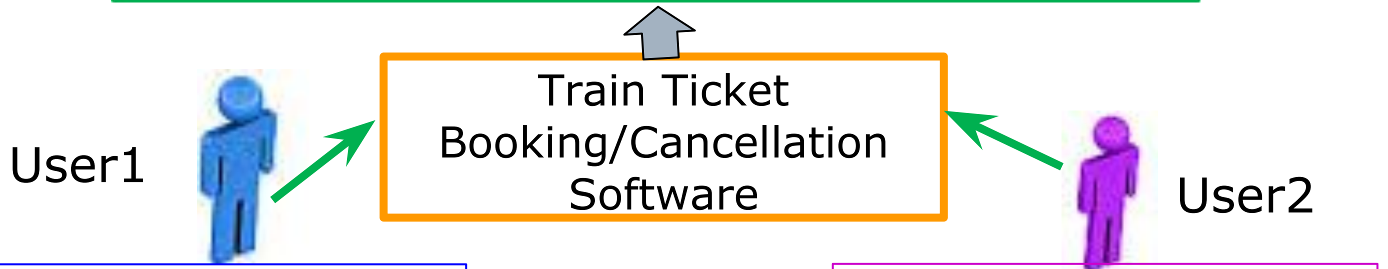
What will be the total number of seats on Tippu express and Chamundi express if first User1 executes Transaction1 and then User2 Executes Transaction2

To Understand problems with concurrent executions



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	~~80~~ 79
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	~~70~~ 75



Transaction1
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)

Transaction2
Read(TippuSeats)
TippuSeats=TippuSeats+4
Write(TippuSeats)

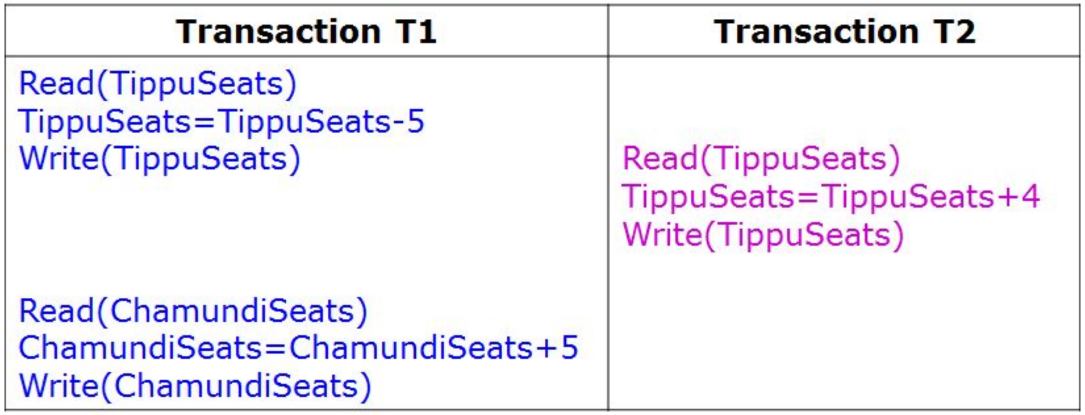
Answer
Tippu express seats: 79
Chamundi express seats: 75

Non-interleaved vs Interleaved transactions

Non-interleaved transaction (or Serial transaction): In this case first completely transaction T1 gets executed and then transaction T2 gets executed

Transaction1
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)

Transaction2
Read(TippuSeats)
TippuSeats=TippuSeats+4
Write(TippuSeats)



Transaction T1	Transaction T2

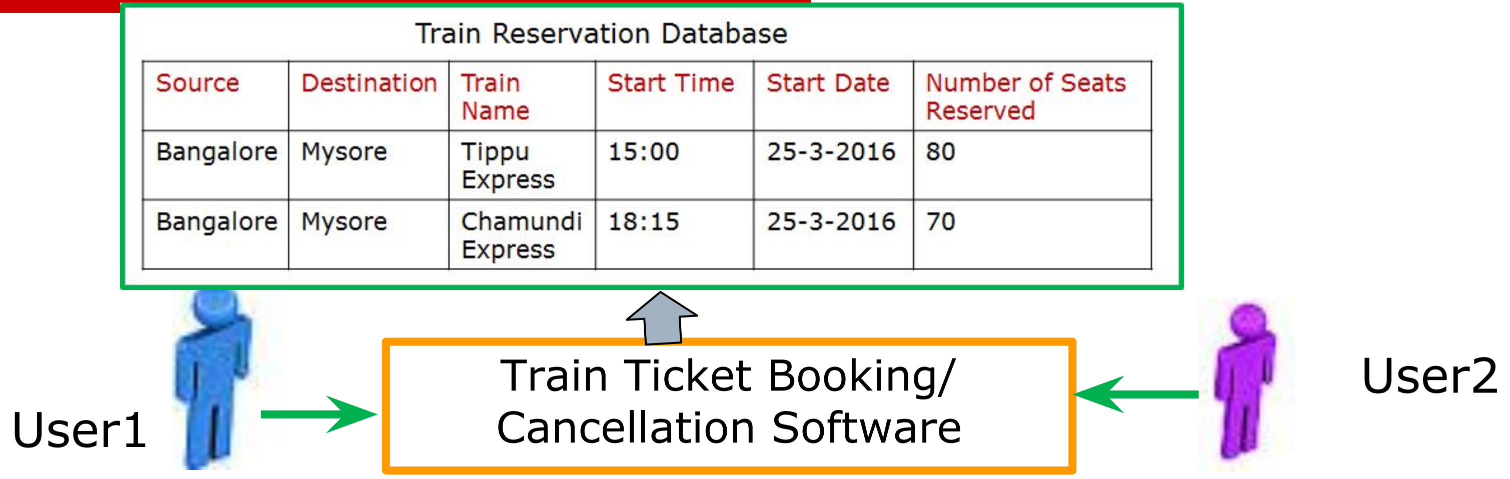
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)	

	Read(TippuSeats)
	TippuSeats=TippuSeats+4
	Write(TippuSeats)

Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Interleaved transaction (or Non-Serial or Concurrent):
First, Part of Transaction T1 gets executed,
second Transaction T2 gets executed and third
remaining part of transaction T1 gets executed

Lost Update Problem: Example



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

Transaction T1	Transaction T2
Read(TippuSeats)	
TippuSeats=TippuSeats-5	
	Read(TippuSeats)
	TippuSeats=TippuSeats+4
Write(TippuSeats)	
Read(ChamundiSeats)	
	Write(TippuSeats)
ChamundiSeats=ChamundiSeats+5	
Write(ChamundiSeats)	

Problems with Concurrent Execution

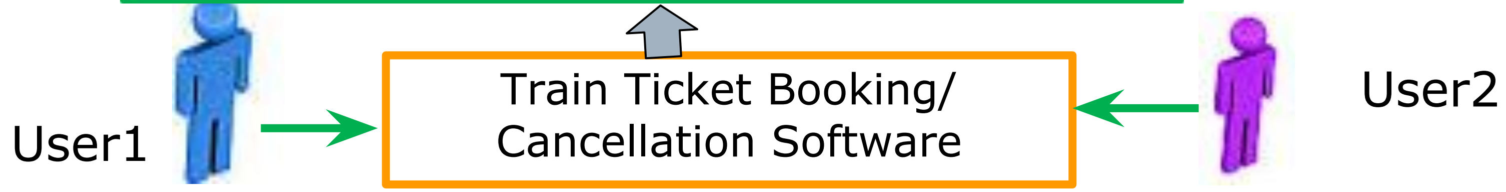
1. Lost Update Problem

This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.

Lost Update Problem: Example

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70



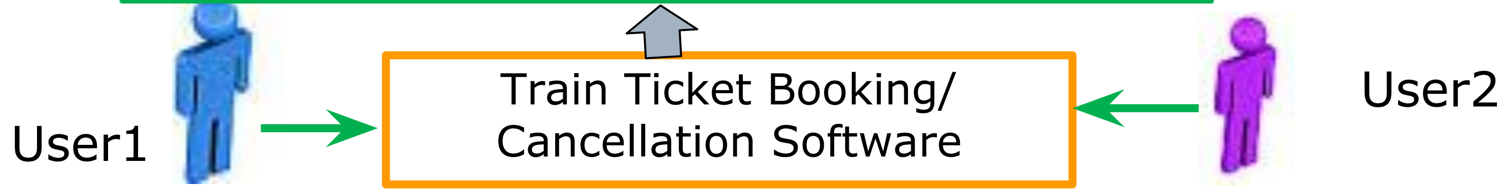
Transaction T1	Transaction T2
Read(TippuSeats)	
TippuSeats=TippuSeats-5	
	Read(TippuSeats)
	TippuSeats=TippuSeats+4
Write(TippuSeats)	
Read(ChamundiSeats)	
	Write(TippuSeats)
ChamundiSeats=ChamundiSeats+5	
Write(ChamundiSeats)	

Question
What will be the total number of seats on Tippu express and Chamundi express after execution of the above set Transaction statements

Lost Update Problem: We are Loosing update

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	~~80~~ 84
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	~~70~~ 75



Transaction T1	Transaction T2

Read(TippuSeats)
TippuSeats=TippuSeats-5	

	Read(TippuSeats)
	TippuSeats=TippuSeats+4

Write(TippuSeats)
Read(ChamundiSeats)	

	Write(TippuSeats)

ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Answer

Tippu express seats: 84 <- **INCORRECT**

Update made by one Transaction is overridden by another Transaction

Chamundi express seats: 75

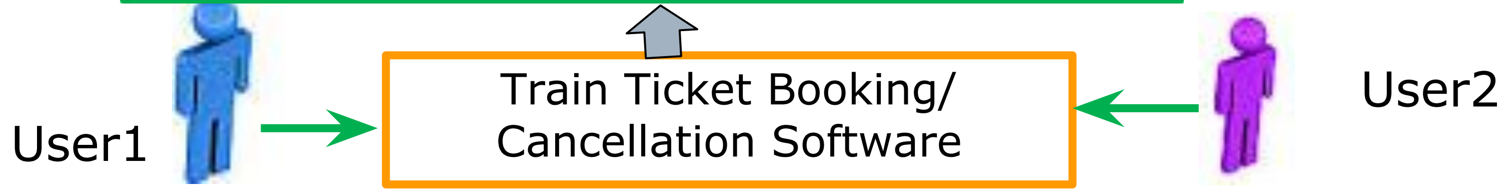
Problems with Concurrent Execution

2. The Dirty Read (or Temporary Update) Problem
   - This occurs when one transaction updates a database item and then the transaction fails for some reason.
   - The updated item is accessed by another transaction before it is changed back to its original value.

Dirty Read (or Temporary Update) Problem

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70



Transaction T1	Transaction T2

Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)	

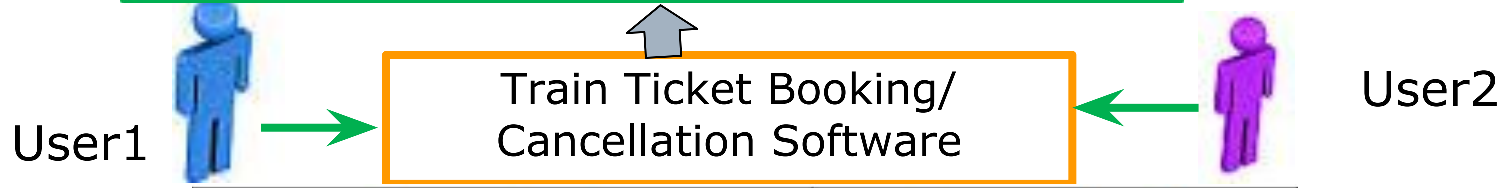
	Read(TippuSeats)
	TippuSeats=TippuSeats+4
	Write(TippuSeats)

UNDO or FAIL or ROLLBack
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Dirty Read (or Temporary Update) Problem

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70



Transaction T1	Transaction T2

Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)	

	Read(TippuSeats)
	TippuSeats=TippuSeats+4
	Write(TippuSeats)

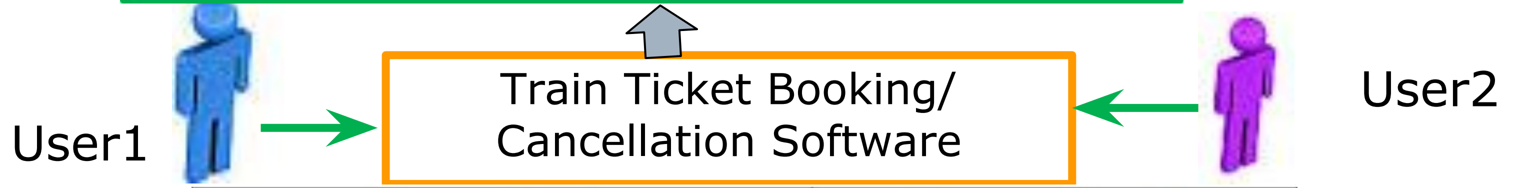
UNDO or FAIL or ROLLBack
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Question
What will be the total number of seats on Tippu express and Chamundi express after execution of the above set Transaction statements

Dirty Read (or Temporary Update) Problem

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70



Transaction T1	Transaction T2
Read(TippuSeats)	
TippuSeats=TippuSeats-5	
Write(TippuSeats)	Read(TippuSeats)
	TippuSeats=TippuSeats+4
	Write(TippuSeats)
UNDO or FAIL or ROLLBack	
Read(ChamundiSeats)	
ChamundiSeats=ChamundiSeats+5	
Write(ChamundiSeats)	

Answer

Tippu express seats: 80 <- **INCORRECT** "Dirty read" / Reading uncommitted data

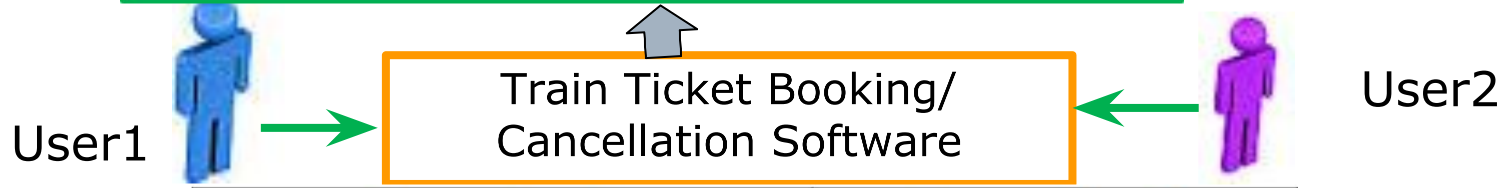
*Occurring with / because of a **write conflict***

Chamundi express seats: 75

Example

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70



Transaction T1	Transaction T2

Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)	

	Read(TippuSeats)
	TippuSeats=TippuSeats+4
	Write(TippuSeats)

UNDO or FAIL or ROLLBack
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Answer

Tippu express seats: 80 <- **INCORRECT** "Dirty read" / Reading uncommitted data
*Occurring with / because of a **write conflict***

Chamundi express seats: 75

Problems with Concurrent Execution

- What is Concurrent Execution ?

When Multiple users trying to access same database record in an uncontrolled manner.

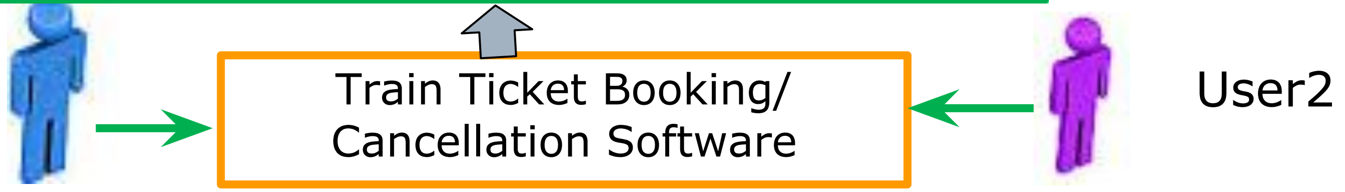
- Problems with Concurrent execution

1. **Lost Update Problem**
2. **Temporary Update (or Dirty Read) Problem**
3. Incorrect Summary Problem
4. Unrepeatable Read

Lost Update Problem: We are Loosing update

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	~~80~~ 84
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	~~70~~ 75



Transaction T1	Transaction T2
Read(TippuSeats)	
TippuSeats=TippuSeats-5	
	Read(TippuSeats)
	TippuSeats=TippuSeats+4
Write(TippuSeats)	
Read(ChamundiSeats)	
	Write(TippuSeats)
ChamundiSeats=ChamundiSeats+5	
Write(ChamundiSeats)	

Answer

Tippu express seats: 84 <- **INCORRECT**

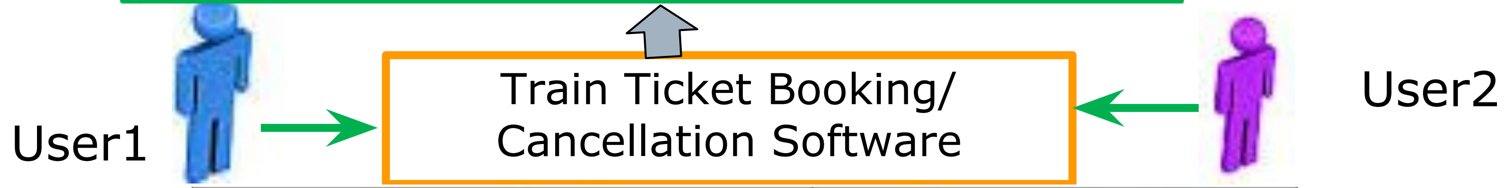
Chamundi express seats: 75

Update made by one Transaction is overridden by another Transaction

Dirty Read (or Temporary Update) Problem

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70



Transaction T1	Transaction T2
Read(TippuSeats)	
TippuSeats=TippuSeats-5	
Write(TippuSeats)	
	Read(TippuSeats)
	TippuSeats=TippuSeats+4
	Write(TippuSeats)
UNDO or FAIL or ROLLBack	
Read(ChamundiSeats)	
ChamundiSeats=ChamundiSeats+5	
Write(ChamundiSeats)	

Answer

Tippu express seats: 80 <- **INCORRECT** "Dirty read" / Reading uncommitted data

*Occurring with / because of a **write conflict***

Chamundi express seats: 75

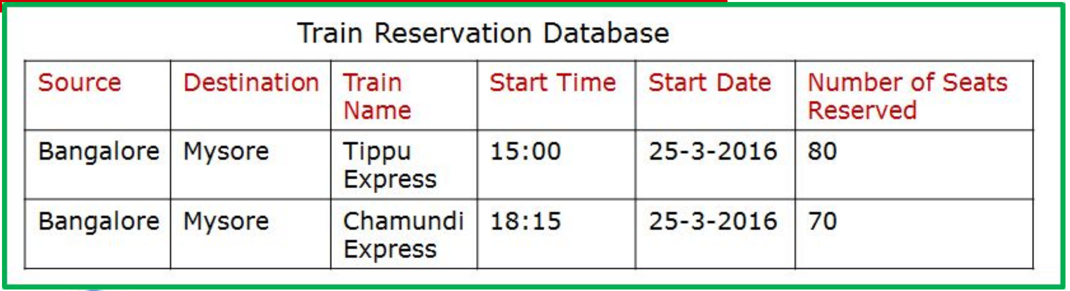
Problems with Concurrent Execution

- What is Concurrent Execution ?

When Multiple users trying to access same database record in an uncontrolled manner.

- Problems with Concurrent execution
  1. Lost Update Problem
  2. Temporary Update (or Dirty Read) Problem
  3. **Incorrect Summary Problem**
  4. Unrepeatable Read

Example



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

User1

Transaction T1

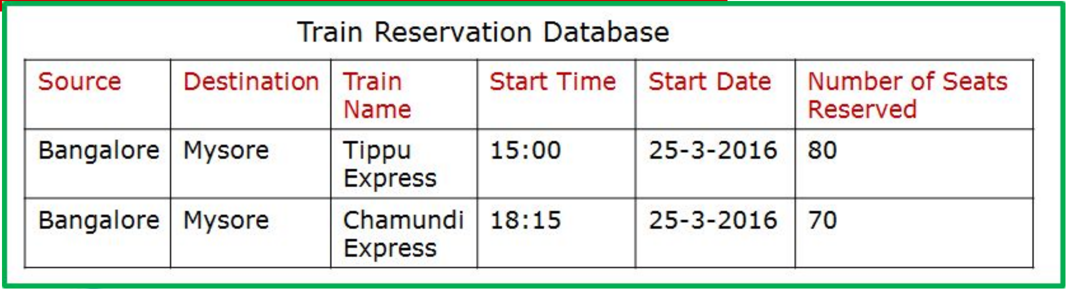
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)

User3

Transaction T3

sum:=0
Read(TippuSeats)
sum=sum+TippuSeats
Read(ChamundiSeats)
sum=sum+ChamundiSeats

Example


Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

User1

Transaction T1

```
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)
```

User3

Transaction T3

```
sum:=0
Read(TippuSeats)
sum=sum+TippuSeats
Read(ChamundiSeats)
sum=sum+ChamundiSeats
```

Question

Say if first Transaction T1 has been executed first and second Transaction T3 has been executed then,
What will be the total number of seats on Tippu express & Chamundi express; and **sum** value

Example

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

User1

Transaction T1

```
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)
```

User3

Transaction T3

```
sum:=0
Read(TippuSeats)
sum=sum+TippuSeats
Read(ChamundiSeats)
sum=sum+ChamundiSeats
```

Answer

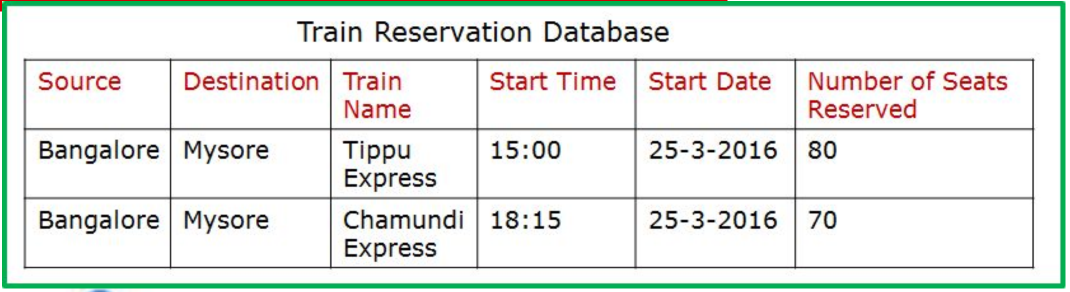
Tippu express seats=75
Chamundi express seats=75
sum=150

Problems with Concurrent Execution

3. The Incorrect Summary Problem

- If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

Incorrect Summary Problem



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

User1

User3

Transaction T1	Transaction T3
	sum:=0

Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)	

	Read(TippuSeats)
	sum=sum+TippuSeats
	Read(ChamundiSeats)
	sum=sum+ChamundiSeats

Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Incorrect Summary Problem

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

User1

User3

Transaction T1	Transaction T3
	sum:=0

Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)	

	Read(TippuSeats)
	sum=sum+TippuSeats
	Read(ChamundiSeats)
	sum=sum+ChamundiSeats

Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Question
What will be the total number of seats on Tippu express & Chamundi express; and sum value when above set of transactions statements are executed

Incorrect Summary Problem

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

User1

User3

Transaction T1	Transaction T3
	sum:=0

Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)	

	Read(TippuSeats)
	sum=sum+TippuSeats
	Read(ChamundiSeats)
	sum=sum+ChamundiSeats

Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Answer

Tippu express seats=75
Chamundi express seats=75
Sum=145 <- **INCORRECT**

Problems with Concurrent Execution

- What is Concurrent Execution ?

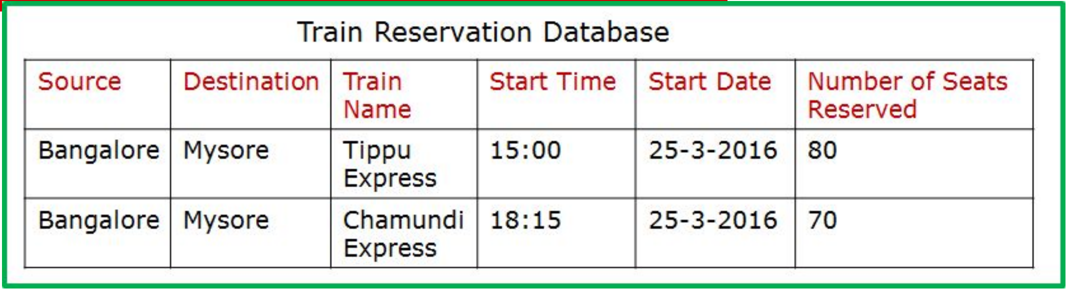
When Multiple users trying to access same database record in an uncontrolled manner.

- Problems with Concurrent execution
  1. Lost Update Problem
  2. Temporary Update (or Dirty Read) Problem
  3. Incorrect Summary Problem
  4. **Unrepeatable Read**

Unrepeatable Read

- Unrepeatable Read occurs, if transaction T1 reads an item twice and the item is changed by an another transaction T2 between two reads hence T1 finds two different values on it's two reads.
- Example: If during train reservation, a user inquires about seat availability on several trains. When user decides on a particular train, the transaction reads the number of seats on that train a second time before completing the reservation.

Unrepeatable Read: Example



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

Transaction T1	Transaction T2
Read(TippuSeats)	

	Read(TippuSeats)
	TippuSeats=TippuSeats+4
	Write(TippuSeats)

Read(TippuSeats)	

First time when Transaction T1 reads, TippuSeats value will be 80 but second time when the same Transaction T1 Reads, TippuSeats value will be 84. T1 is seeing two different values for same item TippuSeats

Why Concurrency Control is needed ?

To avoid following Problems

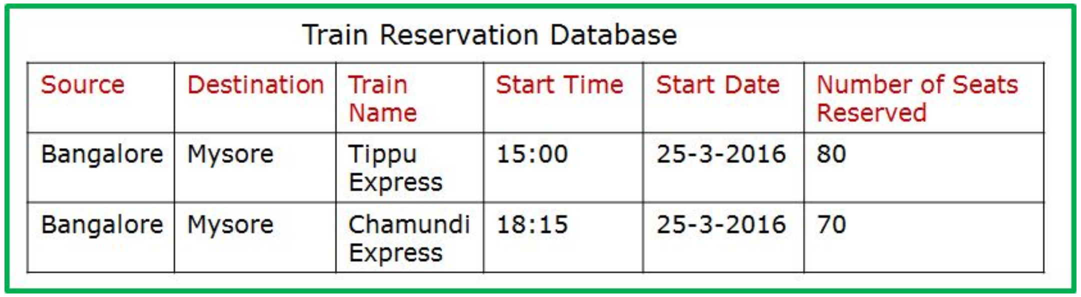
1. Lost Update Problem
2. Temporary Update (or Dirty Read) Problem
3. Incorrect Summary Problem
4. Unrepeatable Read

Next we will Understand Schedules

Schedule is a sequence of operations of various transactions

Transaction Schedules

Example to Understand Transaction Schedules



Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Number of Seats Reserved
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70

Consider TWO transactions **T1** and **T2**

- **T1:** Cancel FIVE seats on Tippu express

Read(TippuSeats)

TippuSeats=TippuSeats-5

Write(TippuSeats)

- **T2:** Reserve FOUR seats on Chamundi express

Read(ChamundiSeats)

ChamundiSeats=ChamundiSeats+4

Write(ChamundiSeats)

Transaction Schedules

Schedule 1: T1, T2

```
T1:Read(TippuSeats)
T1:TippuSeats=TippuSeats-5
T1:Write(TippuSeats)
T2:Read(ChamundiSeats)
T2:ChamundiSeats=ChamundiSeats+4
T2:Write(ChamundiSeats)
```

↓ Shorthand Notation

Schedule 1: T1, T2

```
R1, W1, R2,W2
```

Transaction Schedules

Schedule 1: T1, T2

```
T1:Read(TippuSeats)
T1:TippuSeats=TippuSeats-5
T1:Write(TippuSeats)
T2:Read(ChamundiSeats)
T2:ChamundiSeats=ChamundiSeats+4
T2:Write(ChamundiSeats)
```

Shorthand Notation

Schedule 1: T1, T2

R1, W1, R2,W2

Schedule 2: T2, T1

```
T2:Read(ChamundiSeats)
T2:ChamundiSeats=ChamundiSeats+4
T2:Write(ChamundiSeats)
T1:Read(TippuSeats)
T1:TippuSeats=TippuSeats-5
T1:Write(TippuSeats)
```

Shorthand Notation

Schedule 2: T2, T1

R2,W2, R1, W1

Different Possible Schedules for given set of transactions

Consider Two Transactions
T1: Cancel FIVE seats on Tippu express (R1, W1)
T2: Reserve four seats on Chamundi express (R2,W2)

Different possible schedules for the above two transactions are as follows:

Schedule 1	R1, W1, R2,W2
Schedule 2	R2,W2, R1, W1
Schedule 3	R1, R2, W1,W2
Schedule 4	R2, R1, W2, W1

Transaction Schedules

Consider Two Transactions
T1: Cancel FIVE seats on Tippu express (R1, W1)
T2: Reserve four seats on Chamundi express (R2,W2)

Different possible schedules for the above two transactions are as follows:

Schedule 1	R1, W1, R2,W2
Schedule 2	R2,W2, R1, W1
Schedule 3	R1, R2, W1,,W2
Schedule 4	R2, R1, W2, W1

Following are **not feasible schedules** because they do not preserve the order of operations of the individual Transactions

W1, R1, W2, R2

W2, R2, W1, R1

Transaction Schedules

Consider Two Transactions
T1: Cancel FIVE seats on Tippu express (R1, W1)
T2: Reserve four seats on Chamundi express (R2,W2)

Different possible schedules for the above two transactions are as follows:

Schedule 1	R1, W1, R2,W2
Schedule 2	R2,W2, R1, W1
Schedule 3	R1, R2, W1,,W2
Schedule 4	R2, R1, W2, W1

Following are **not feasible schedules** because they do not preserve the order of operations of the individual Transactions

W1, R1, W2, R2	W2, R2, W1, R1
W1, W2, R1, R2	W2, W1, R2, R1

Serial Schedule vs Interleaved Schedule

Consider Two Transactions
T1: Cancel FIVE seats on Tippu express
T2: Reserve four seats on Chamundi express
Different possible schedules for the above two transactions are as follows:

Schedule 1	R1, W1, R2,W2
Schedule 2	R2,W2, R1, W1
Schedule 3	R1, R2, W1,W2
Schedule 4	R2, R1, W2, W1

Schedule 1 & 2 are Serial Schedule

Schedule 3 & 4 are Interleaved or Parallel Schedule

Note:
Serial Schedule: A schedule in which the different transactions are NOT interleaved (i.e., transactions are executed from start to finish one-by-one)

-The result of **Interleaved Schedule** should be equivalent Some serial schedule

Question

- What will be the total number of **serial schedules** that can occur if there are m Transactions i.e., T1, T2,….Tm

Question

- What will be the total number of **serial schedules** that can occur if there are m Transactions i.e., T1, T2,….Tm
- Answer: **m!** (m * (m-1) * (m-2)…..*1)

Example:

- Two Transactions T1 & T2

Two (2!) Serial Schedules:

T1, T2

T2,T1

- Three Transactions T1 , T2 & T3

Six (3!) Serial Schedules :

T1,T2,T3

T1,T3,T2

T2,T1,T3

T2,T3,T1

T3,T1,T2

T3,T2,T1

Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions

- A and B are **equivalent schedules** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B

- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word "**some**" makes this definition powerful and tricky!

Problem to Solve

To check whether given schedule is **serializable or not**

Example- consider two TXNs:

```
T1: START TRANSACTION
      UPDATE Accounts
      SET Amt = Amt + 100
      WHERE Name = ‘A’

      UPDATE Accounts
      SET Amt = Amt - 100
      WHERE Name = ‘B’
COMMIT
```

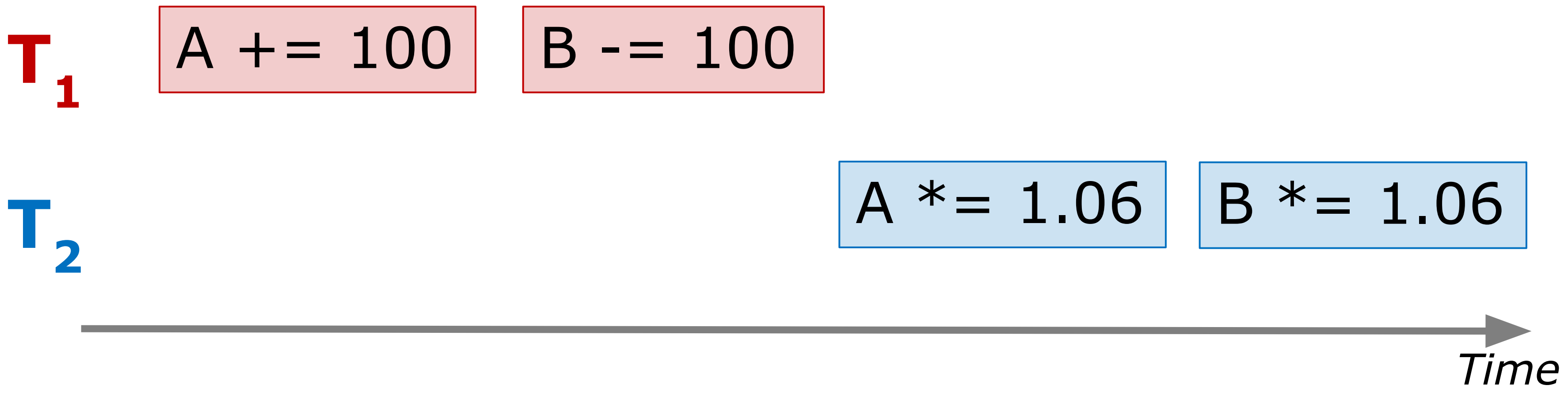
```
T2: START TRANSACTION
      UPDATE Accounts
      SET Amt = Amt * 1.06
COMMIT
```

T1 transfers Rs.100/- from B’s account to A’s account

T2 credits both accounts with a 6% interest payment

Example- consider two Transactions (T1 and T2):

We can look at the transactions in a timeline view- serial execution:

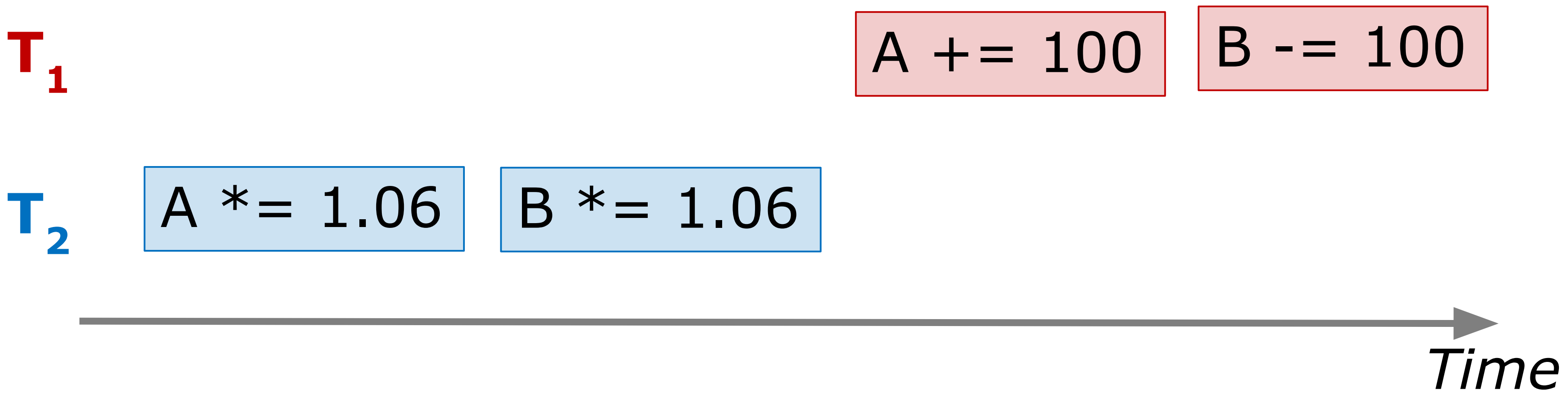


T1 transfers Rs.100 from B’s account to A’s account

T2 credits both accounts with a 6% interest payment

Example- consider two Transactions (T1 and T2):

The transactions could occur in either order… DBMS allows!



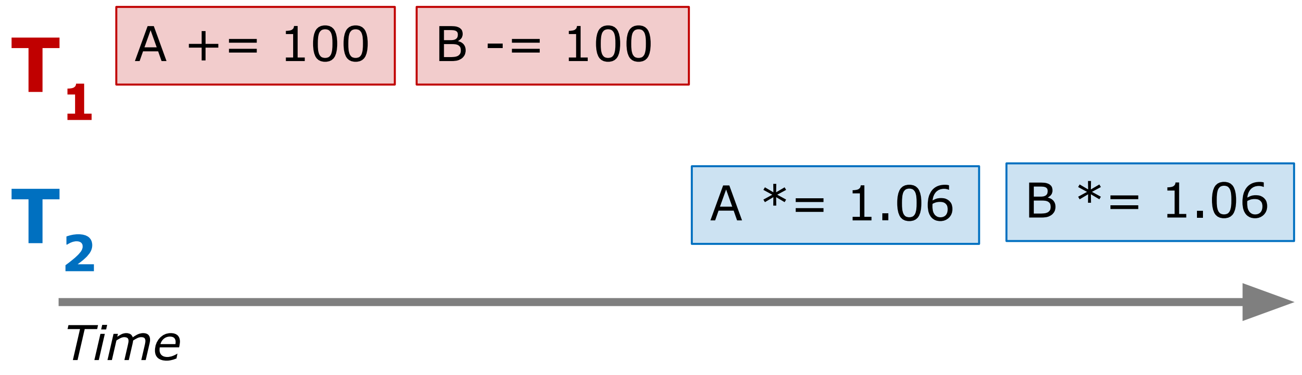
T2 credits both accounts with a 6% interest payment

T1 transfers Rs.100/- from B's account to A's account

Serial schedule T_1,T_2:

Starting Balance

A	B
Rs.50	Rs.200



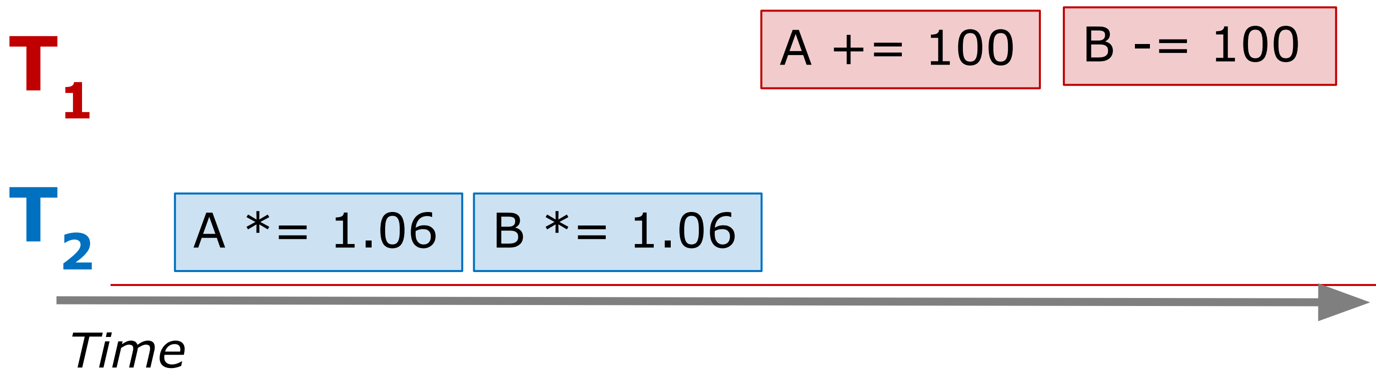
Result of Executing T1, T2

A	B
Rs.159	Rs.106

Serial schedule T_2,T1:

Starting Balance

A	B
Rs.50	Rs.200



Result of Executing T2,T1

A	B
Rs.153	Rs.112

Problem to Solve

Check whether the following schedule i.e., (A+=100, A*=1.06, B-=100, B*=1.06) is Serializable ?

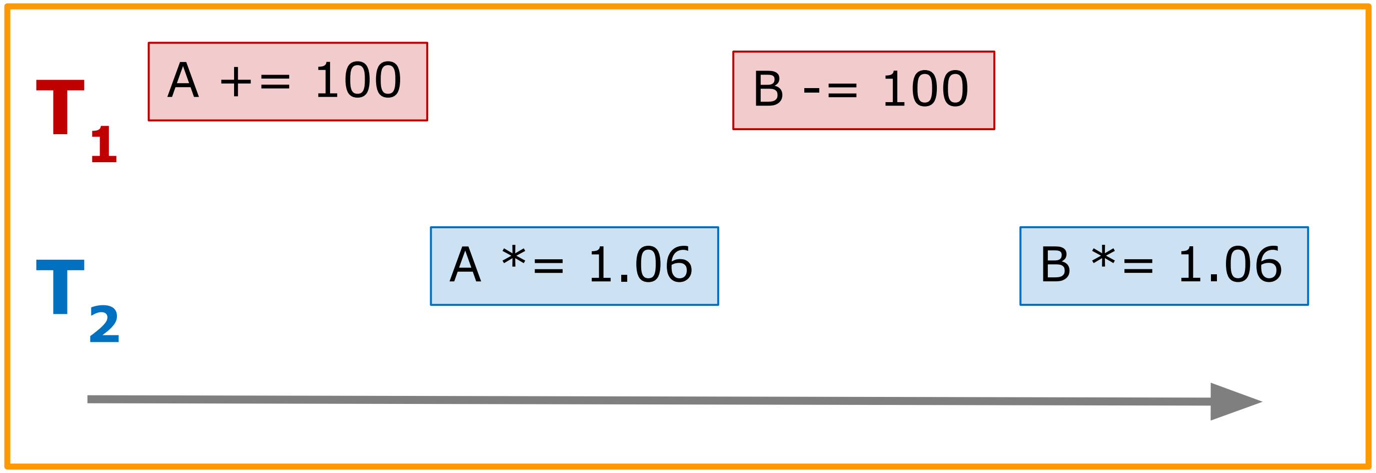
Starting Balance

A	B
Rs.50	Rs.200

Serial schedule results:

	A	B
T_1,T_2	Rs.159	Rs.106
T_2,T_1	Rs.153	Rs.112

Transaction Schedule



Note:To check whether given schedule is serializable or not, we should check whether the given interleaved schedule result is equivalent to result of some serial schedule

Serializable, **Yes**

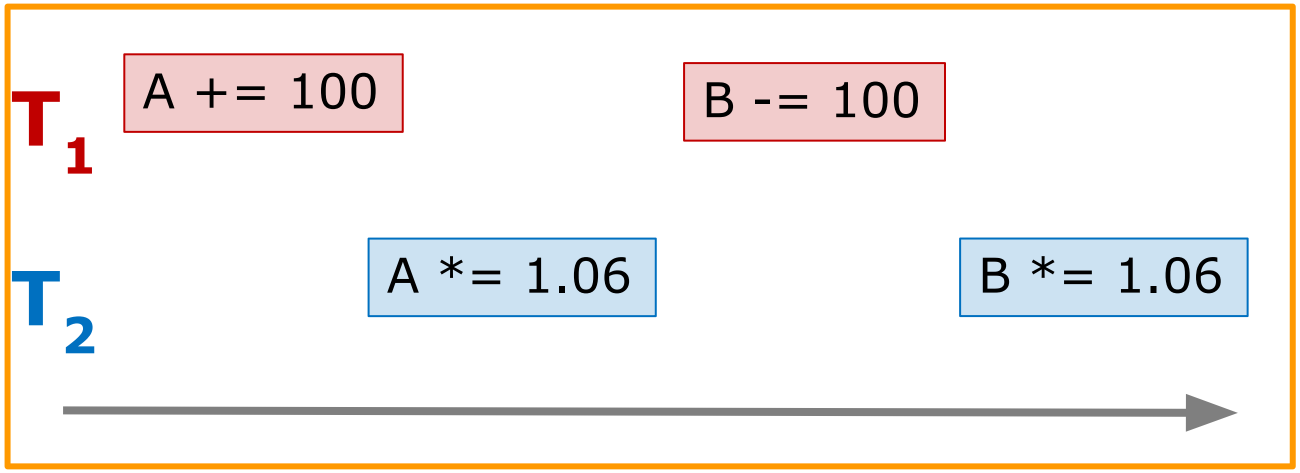
Starting Balance

A	**B**
Rs.50	Rs.200

Serial schedules:

	A	**B**
T_1,T_2	Rs.159	Rs.106
T_2,T_1	Rs.153	Rs.112

Transaction Schedule



A	**B**
Rs.159	Rs.106

Same as a serial schedule ***for all possible values of A, B*** **=** <u>**serializable**</u>

Problem to Solve

Check whether the following schedule i.e., (A+=100, A*=1.06, B-=100, B*=1.06) is Serializable ?

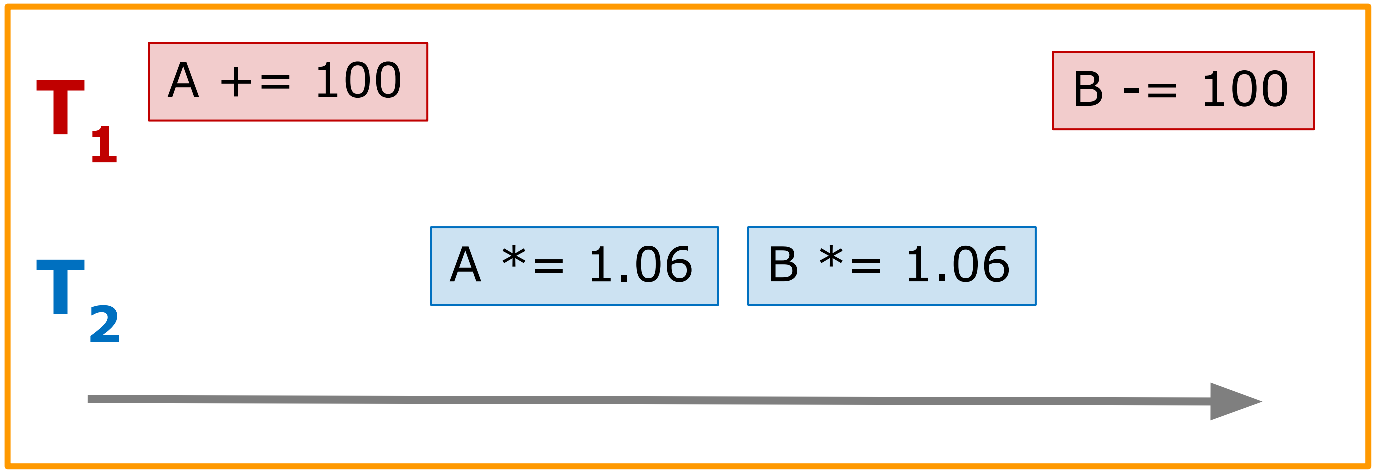
Starting Balance

A	B
Rs.50	Rs.200

Serial schedule results:

	A	B
T_1,T_2	Rs.159	Rs.106
T_2,T_1	Rs.153	Rs.112

Transaction Schedule



Note:To check whether given schedule is serializable or not, we should check whether the given interleaved schedule result is equivalent to result of some serial schedule

Serializable, **No**

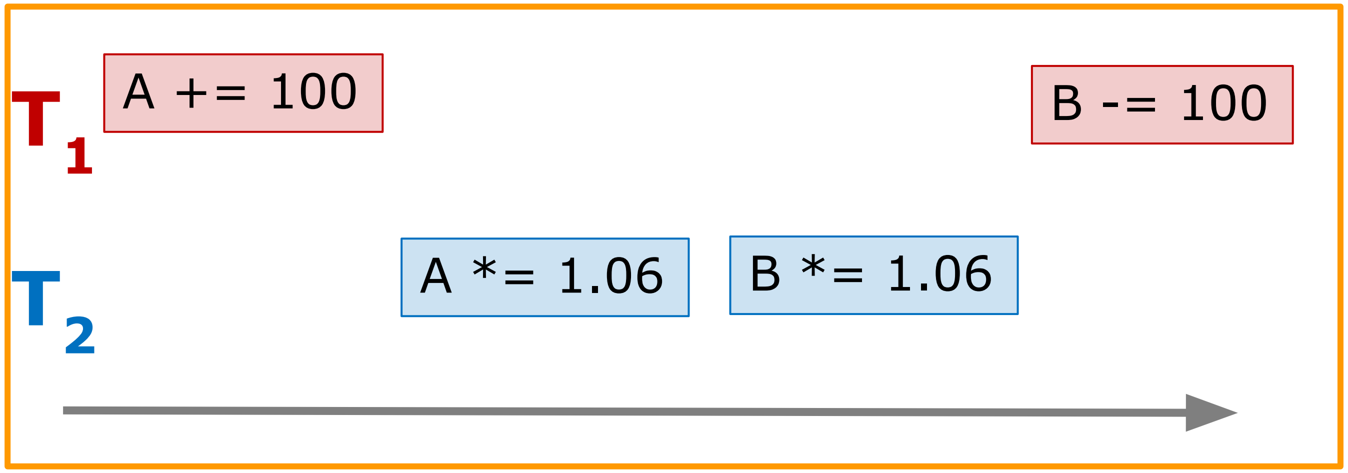
Starting Balance

A	B
Rs.50	Rs.200

Serial schedules:

	A	B
T_1,T_2	Rs.159	Rs.106
T_2,T_1	Rs.153	Rs.112

Transaction Schedule

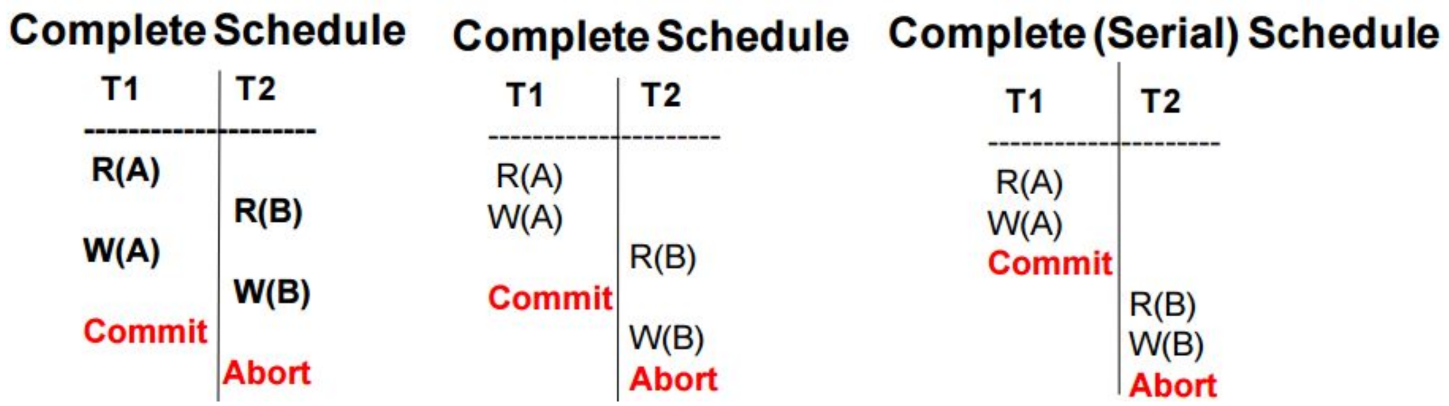


A	B
Rs.159	Rs.112

Not *equivalent* to any serializable schedule **=** ***not*** **<u>serializable</u>**

Complete Schedule

- Complete Schedule: A schedule that **contains either a commit or an abort** action for EACH transaction



Complete Schedule

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

Complete Schedule

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

Complete (Serial) Schedule

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

Note: consequently, a complete schedule will not contain any active transactions at the end of the schedule

Next we will Understand

Conflicting operations in Schedules

Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy **all three** of the following **conditions**

1. If two operations belong to different transactions
2. If two operations access same data item
3. Among two operations at least one operation is write

Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy all three of the following conditions

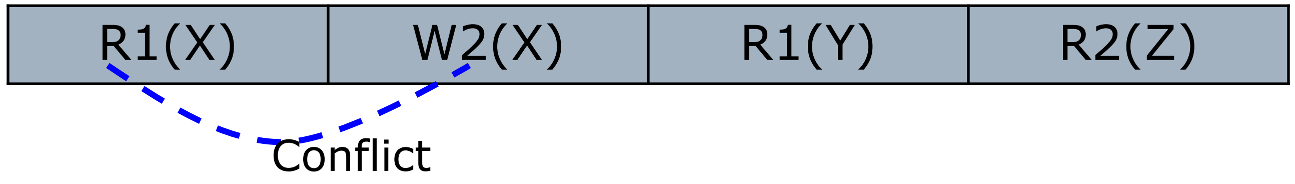
1. If two operations belong to **different transactions**
2. If two operations access same data item
3. Among two operations at least one operation is write

Example: Consider two transactions

T1 with operations R1(X) and R1(Y)

T2 with operations W2(X) and R2(Z)

Consider the schedule as: R1(X), W2(X), R1(Y), R2(Z)



Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy all three of the following conditions

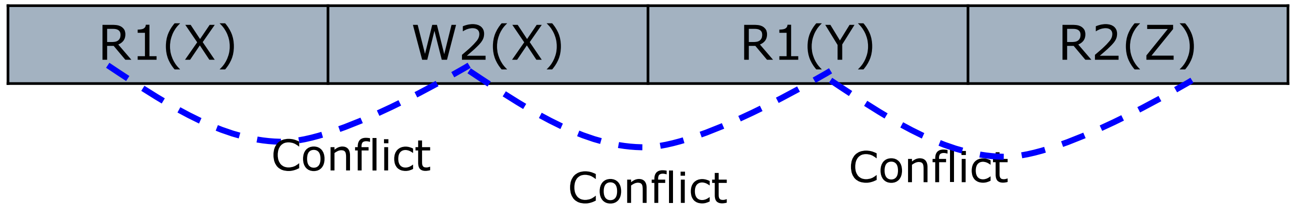
1. If two operations belong to **different transactions**
2. If two operations access same data item
3. Among two operations at least one operation is write

Example: Consider two transactions

T1 with operations R1(X) and R1(Y)

T2 with operations W2(X) and R2(Z)

Consider the schedule as: R1(X), W2(X), R1(Y), R2(Z)



Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy all three of the following conditions

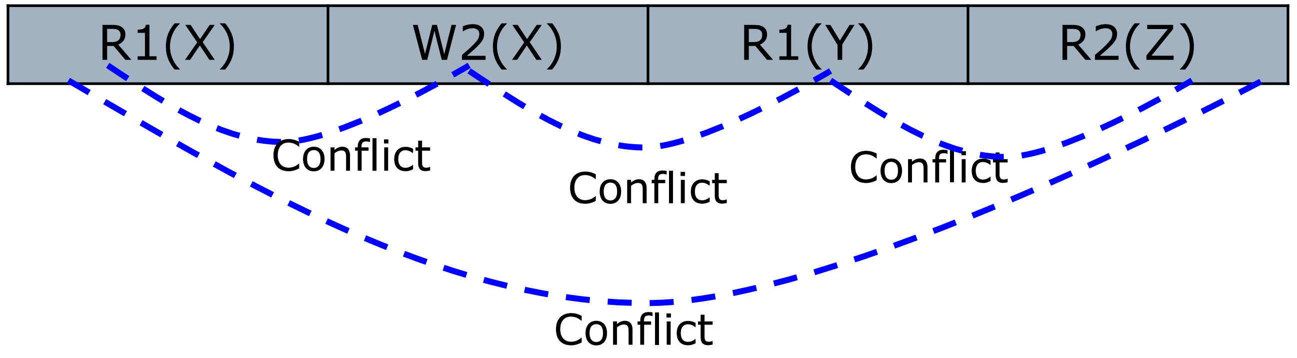
1. If two operations belong to **different transactions**
2. If two operations access same data item
3. Among two operations at least one operation is write

Example: Consider two transactions

T1 with operations R1(X) and R1(Y)

T2 with operations W2(X) and R2(Z)

Consider the schedule as: R1(X), W2(X), R1(Y), R2(Z)



Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy all three of the following conditions

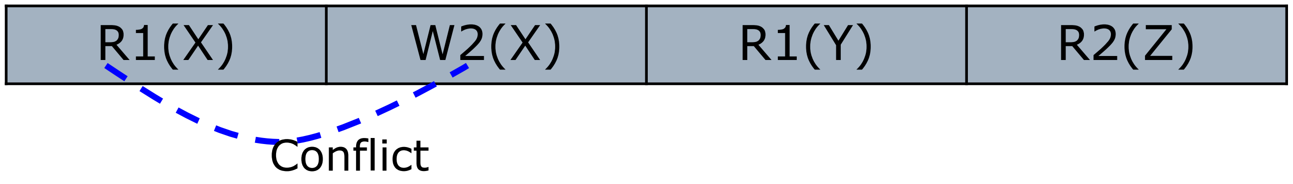
1. If two operations belong to different transactions
2. If two operations **access same data item**
3. Among two operations at least one operation is write

Example: Consider two transactions

T1 with operations R1(X) and R1(Y)

T2 with operations W2(X) and R2(Z)

Consider the schedule as: R1(X), W2(X), R1(Y), R2(Z)



Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy all three of the following conditions

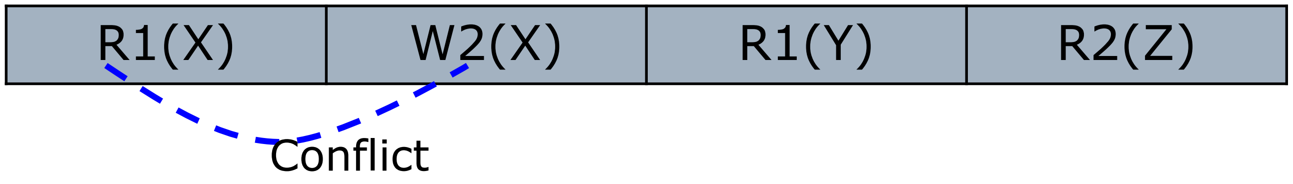
1. If two operations belong to different transactions
2. If two operations access same data item
3. Among two operations **at least one operation is write**

Example: Consider two transactions

T1 with operations R1(X) and R1(Y)

T2 with operations W2(X) and R2(Z)

Consider the schedule as: R1(X), W2(X), R1(Y), R2(Z)



Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy **all three of the following conditions**

1. If two operations belong to different transactions
2. If two operations access same data item
3. Among two operations at least one operation is write

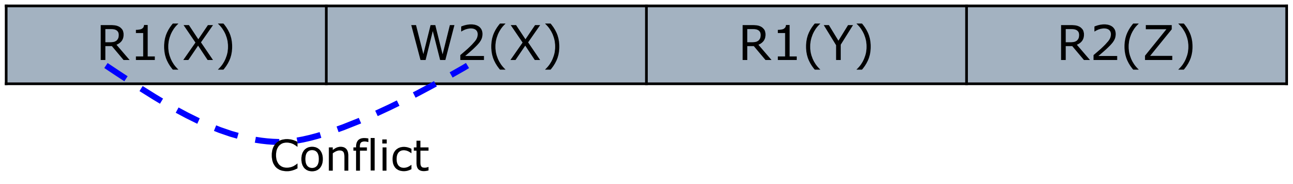
Example: Consider two transactions

T1 with operations R1(X) and R1(Y)

T2 with operations W2(X) and R2(Z)

Consider the schedule as: R1(X), W2(X), R1(Y), R2(Z)

Now only CONFLICTing operation in the given schedule which satisfy all three conditions is **(R1(X), W2(X))**



Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy all three of the following conditions

1. If two operations belong to different transactions
2. If two operations access same data item
3. Among two operations at least one operation is write

Question:
Check which of the following schedules are having Conflicting operations

Schedule 1	R1(X), W2(X), W3(X)	
Schedule 2	R1(X), R2(X), R3(X)	
Schedule 3	R1(X), W2(Y), R3(X)	

Conflicting Operations in Schedules

Two operations in schedule are said to CONFLICT if they satisfy all three of the following conditions

1. If two operations belong to different transactions
2. If two operations access same data item
3. Among two operations at least one operation is write

Question:
Check which of the following schedules are having Conflicting operations

Schedule 1	R1(X), W2(X), W3(X)	YES, because (R1(X), W2(X))
Schedule 2	R1(X), R2(X), R3(X)	NO
Schedule 3	R1(X), W2(Y), R3(X)	NO

Serializibility

What is Serializability ? – “Correctness Measure” of some Schedule

– Why is it useful? It answers the question: “Will an interleaved schedule execute correctly”

– i.e., a Serializable schedule will execute as correctly as serial schedule … but in an interleaved manner!

Example: Consider two transactions

T1 with operations

Read(X)

X=X-5

Write(X)

Read(Y)

Y=Y+5

Write(Y))

T2 with operations

Read(X)

X=X-4

Write(X)

Serializibility

What is Serializability ? – “Correctness Measure” of some Schedule

– Why is it useful? It answers the question: “Will an interleaved schedule execute correctly”

– i.e., a Serializable schedule will execute as correctly as serial schedule … but in an interleaved manner!

Example: Consider two transactions

T1 with operations (Read(X); X=X-5,Write(X); Read(Y); Y=Y+5; Write(Y))

T2 with operations (Read(X), X=X-4,Write(X));

For this two Transactions (T1 & T2) two possible serial schedules are:

Serial schedule **(T1, T2)**

Order of Execution Of operations ↓

Transaction T1	Transaction T2
Read(X)	
X=X-5	
Write(X)	
Read(Y)	
Y=Y+5	
Write(Y)	
	Read(X)
	X=X+4
	Write(X)

Serial schedule **(T2, T1)**

Transaction T1	Transaction T2
	Read(X)
	X=X+4
	Write(X)
Read(X)	
X=X-5	
Write(X)	
Read(Y)	
Y=Y+5	
Write(Y)	

Order of Execution Of operations ↓

Serializibility

Serial schedule (T1, T2)

Order of Execution Of operations

Transaction T1	Transaction T2
Read(X)	
X=X-5	
Write(X)	
Read(Y)	
Y=Y+5	
Write(Y)	
	Read(X)
	X=X+4
	Write(X)

Serial schedule (T2, T1)

Order of Execution Of operations

Transaction T1	Transaction T2
	Read(X)
	X=X+4
	Write(X)
Read(X)	
X=X-5	
Write(X)	
Read(Y)	
Y=Y+5	
Write(Y)	

Non-Serial schedule which is **serializable** because it is equivalent to serial schedule (T1,T2)

Order of Execution Of operations

Transaction T1	Transaction T2
Read(X)	
X=X-5	
Write(X)	
	Read(X)
	X=X+4
	Write(X)
Read(Y)	
Y=Y+5	
Write(Y)	

Serializibility

Serial schedule (T1, T2)

Order of Execution Of operations

Transaction T1	Transaction T2
Read(X)	
X=X-5	
Write(X)	
Read(Y)	
Y=Y+5	
Write(Y)	
	Read(X)
	X=X+4
	Write(X)

Serial schedule (T2, T1)

Transaction T1	Transaction T2
	Read(X)
	X=X+4
	Write(X)
Read(X)	
X=X-5	
Write(X)	
Read(Y)	
Y=Y+5	
Write(Y)	

Order of Execution Of operations

Non-Serial schedule which is **serializable** because it is equivalent to serial schedule (T1,T2)

Order of Execution Of operations

Transaction T1	Transaction T2
Read(X)	
X=X-5	
Write(X)	
	Read(X)
	X=X+4
	Write(X)
Read(Y)	
Y=Y+5	
Write(Y)	

Non-Serial schedule, but it is **not serializable** because it is not equivalent to any serial schedule

Transaction T1	Transaction T2
Read(X)	
X=X-5	
	Read(X)
	X=X+4
Write(X)	
Read(Y)	
	Write(X)
Y=Y+5	
Write(Y)	

Order of Execution Of operations

Characterizing Schedules based on Serializibility

Based on Serializability

Characterize which schedules are correct when concurrent transactions are executing.

1. Conflict Serializable Schedule
2. View Serializable Schedule

Conflict Serializibility Schedule

- A schedule **S** is said to be conflict serializable if it is **conflict equivalent** to some serial schedule **S'**.

What is Conflict Equivalent ?

- Two schedules are said to be conflict equivalent if the order of any two **conflicting operations** is the same in both schedules.

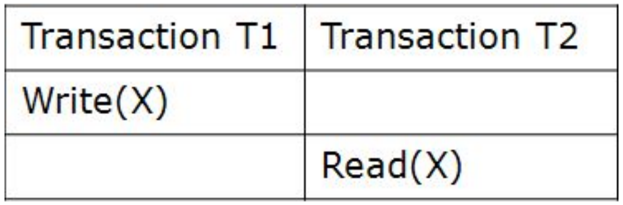
Conflict Serializibility Schedule

- A schedule **S** is said to be conflict serializable if it is **conflict equivalent** to some serial schedule **S'**.

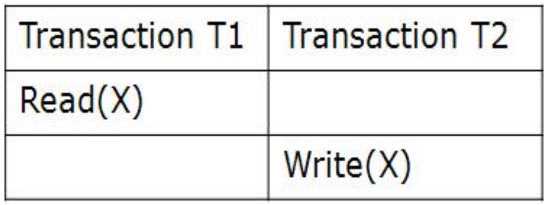
What is Conflict Equivalent ?

- Two schedules are said to be conflict equivalent if the order of any two **conflicting operations** is the same in both schedules.

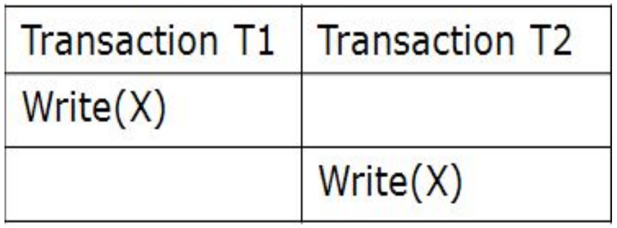
What are Conflicting Operations ?



Transaction T1	Transaction T2
Write(X)	
	Read(X)



Transaction T1	Transaction T2
Read(X)	
	Write(X)



Transaction T1	Transaction T2
Write(X)	
	Write(X)

Check whether the given Two schedules are conflict equivalent ?

- Definition: Two schedules are said to be conflict equivalent if the order of any two **conflicting operations** is the same in both schedules.

Schedule S1

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

Schedule S2

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

Schedule S1: R1(A), R1(B), W2(A), W2(B)
Schedule S2: R1(A),W2(A), R1(B),W2(B)

Check whether the given Two schedules are conflict equivalent ?

Schedule S1

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

Schedule S2

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

Answer: Schedule S1 and S2 are conflict equivalent

Schedule 1	R1(A), R1(B), W2(A),W2(B)	Conflict Operations
		R1(A) and W2(A)
		R1(B) and W2(B)
			Schedule 1 and 2 are conflict equivalent because the order of conflict operations are same

Schedule 2	R1(A),W2(A), R1(B),W2(B)	Conflict Operations
		R1(A) and W2(A)
		R1(B) and W2(B)	

Check whether the given Two schedules are conflict equivalent ?

- Definition: Two schedules are said to be conflict equivalent if the order of any two **conflicting operations** is the same in both schedules.

Schedule S1

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

Schedule S2

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

Schedule S1: W2(A), W2(B), R1(A), R1(B)
Schedule S2: R1(A),W2(A), R1(B),W2(B)

Check whether the given Two schedules are conflict equivalent ?

Schedule S1

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

Schedule S2

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

Answer: Schedule S1 and S2 **are not** conflict equivalent

Schedule 1	W2(A),W2(B), R1(A), R1(B)	Conflict Operations
		W2(A) and **R1(A)**
		W2(B) and R1(B)
			Schedule 1 and 2 are not conflict equivalent because the order of conflict operations are different in the schedules

Schedule 2	R1(A),W2(A), R1(B),W2(B)	Conflict Operations
		R1(A) and W2(A)
		R1(B) and W2(B)	

Problem to Solve

Check whether the following two schedules are conflict equivalent ?

Schedule 1

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

Schedule 2

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

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

Schedule 1	R1(A), W1(A), R2(A),W2(A)	Conflict Operations
		R1(A) and W2(A)
		W1(A) and R2(A)
		W1(A) and W2(A)
			Schedule 1 and 2 are not conflict equivalent because the order of conflict operations are not same

Schedule 2	R2(A), W2(A),R1(A), W1(A)	Conflict Operations
		W2(A) and R1(A)
		R2(A) and W1(A)
		W2(A) and W1(A)	

Note:Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Problem to Solve

Check whether the following two schedules are conflict equivalent ?

Schedule 1

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

Schedule 3

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

Schedule 1	R1(A), W1(A), R2(A),W2(A)	Conflict Operations	
Schedule 3	R1(A), R1(A), W2(A),W1(A)	Conflict Operations	

Note:Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Testing for conflict-serializability of a schedule

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a **precedence graph** (serialization graph) - a graph with directed edges
  - An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles

Definition of Conflict Serializability Schedule:
A schedule **S** is said to be conflict serializable if it is **conflict equivalent** to some serial schedule **S'**.

Algorithm Testing for conflict-serializability

Algorithm:

1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
2. For each case in S where **Tj** executes a **read_item(X)** after **Ti** executes a **write_item(X)**, create an edge (**Ti□Tj**) in the precedence graph.
3. For each case in S where **Tj** executes a **write_item(X)** after **Ti** executes a **read_itern (X)** ,create an edge (**Ti□Tj**) in the precedence graph.
4. For each case in S where **Tj** executes a **write_item(X)** after **Ti** executes a **write_item(X)**, create an edge (**Ti□Tj**) in the precedence graph.
5. The schedule S is **serializable** if and only if the precedence graph has **no cycles**.

Problem To Solve

- Construct precedence graph for the following schedule

Schedule S1

T1	T2	T3
		Read(A)
	Read(A)	
		Write(A)
Read(A)		
Write(A)		

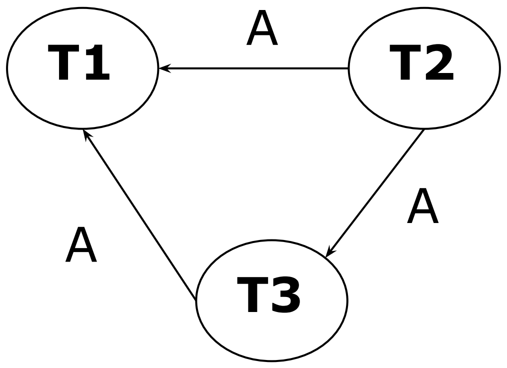
Solution

- Construct precedence graph for the following schedule

Schedule S1

T1	T2	T3
		Read(A)
	Read(A)	
		Write(A)
Read(A)		
Write(A)		

Given: Three Transactions
T1 with operations Read(A), Write(A)
T2 with operations Read(A)
T3 with operations Read(A), Write(A)
Schedule S1: R3(A), R2(A),W3(A), R1(A), W1(A)



$$\begin{tikzcd} T1 & & T2 \arrow[ll, "A"'] \arrow[dl, "A"] \\ & T3 \arrow[ul, "A"] & \end{tikzcd}$$

Problem To Solve

- Construct precedence graph for the following schedule

Schedule S1

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		
		Write(A)

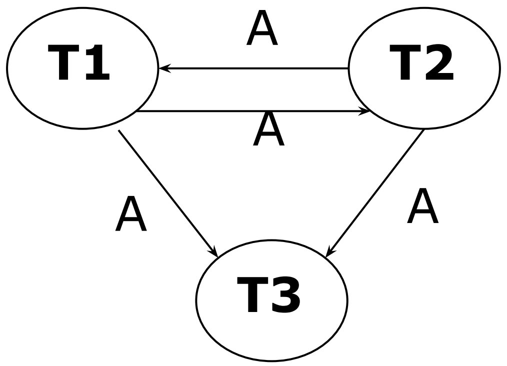
Problem To Solve

- Construct precedence graph for the following schedule

Schedule S1

T1	T2	T3
Read(A)		
	Write(A)	
Write(A)		
		Write(A)

Given: Three Transactions
T1 with operations Read(A), Write(A)
T2 with operations Read(A)
T3 with operations Write(A)
Schedule S1: R1(A), W2(A), W1(A), W3(A)


$$\begin{tikzcd}
T1 \arrow[rr, bend right, "A"'] \arrow[dr, "A"'] & & T2 \arrow[ll, bend right, "A"'] \arrow[dl, "A"] \\
& T3 &
\end{tikzcd}$$

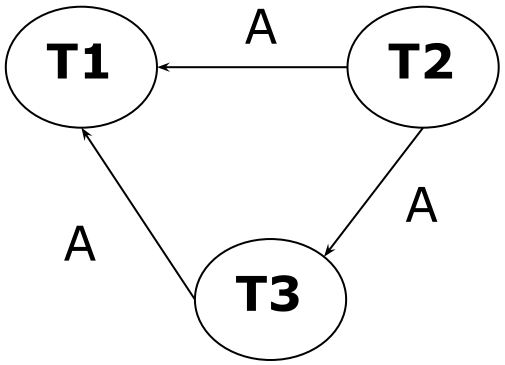
Problem To Solve

- What is the Equivalent serial schedule for non-serial schedule S1:R3(A), R2(A),W3(A), R1(A), W1(A)

Problem To Solve

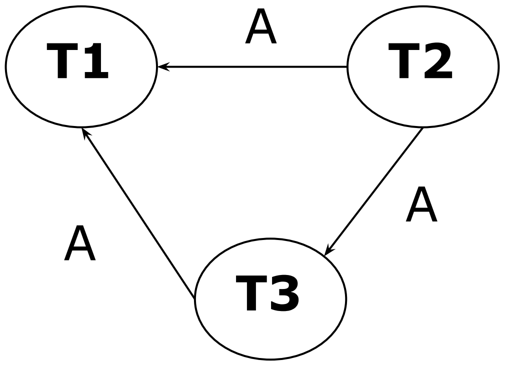
- What is the Equivalent serial schedule for non-serial scheduleS1:R3(A), R2(A),W3(A), R1(A), W1(A)

Given: Three Transactions
T1 with operations Read(A), Write(A)
T2 with operations Read(A)
T3 with operations Read(A), Write(A)
Schedule S1: R3(A), R2(A),W3(A), R1(A), W1(A)

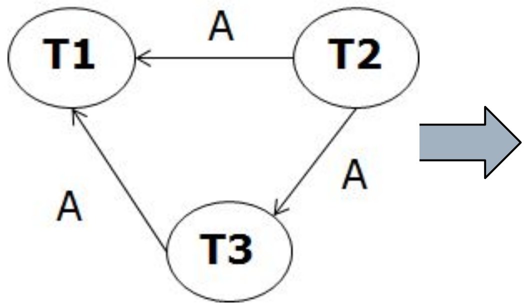

$$\begin{tikzcd} T1 & & T2 \arrow[ll, "A"'] \arrow[dl, "A"] \\ & T3 \arrow[ul, "A"] & \end{tikzcd}$$

Problem To Solve

- What is the Equivalent serial schedule for non-serial scheduleS1:R3(A), R2(A),W3(A), R1(A), W1(A)



Given: Three Transactions
T1 with operations Read(A), Write(A)
T2 with operations Read(A)
T3 with operations Read(A), Write(A)
Schedule S1: R3(A), R2(A),W3(A), R1(A), W1(A)

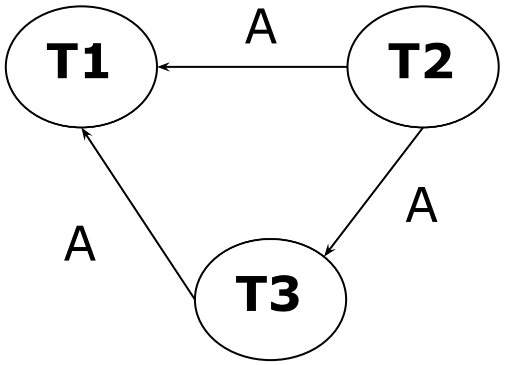


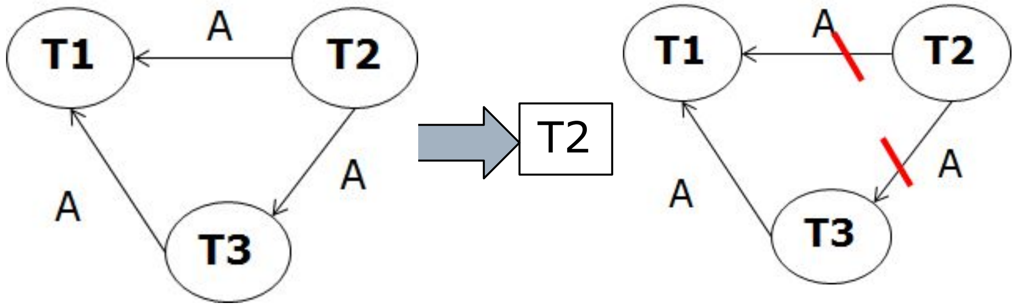
Consider first T2
Because indegree is
one

Problem To Solve

- What is the Equivalent serial schedule for non-serial scheduleS1:R3(A), R2(A),W3(A), R1(A), W1(A)

Given: Three Transactions
T1 with operations Read(A), Write(A)
T2 with operations Read(A)
T3 with operations Read(A), Write(A)
Schedule S1: R3(A), R2(A),W3(A), R1(A), W1(A)

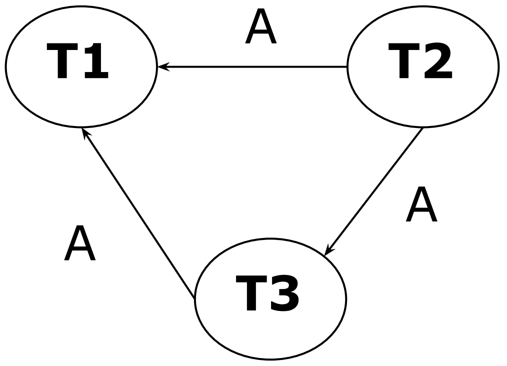




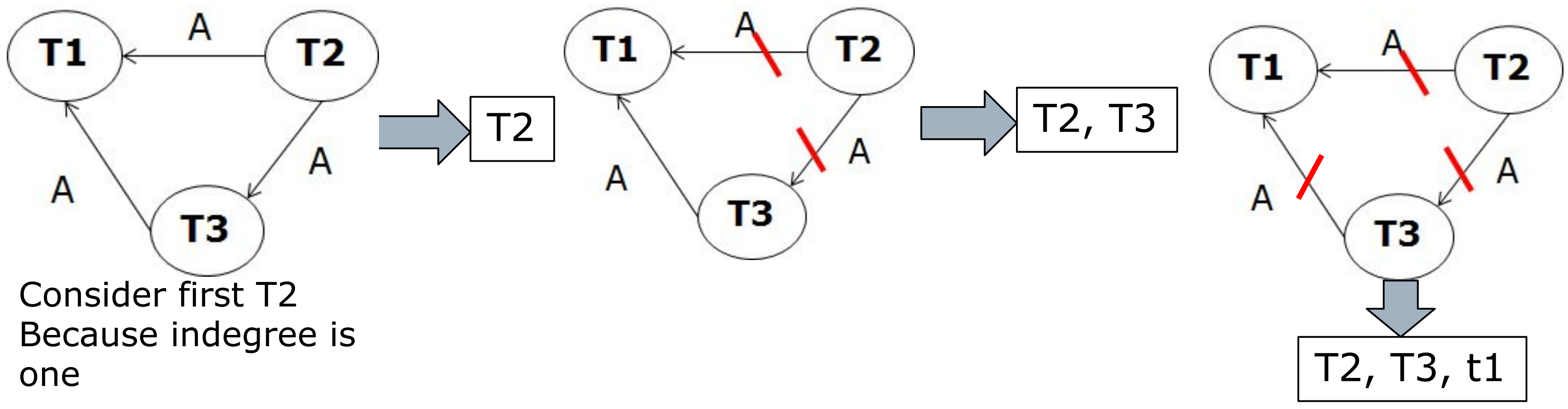
Consider first T2
Because indegree is
one

Problem To Solve

- What is the Equivalent serial schedule for non-serial scheduleS1:R3(A), R2(A),W3(A), R1(A), W1(A)



Given: Three Transactions
T1 with operations Read(A), Write(A)
T2 with operations Read(A)
T3 with operations Read(A), Write(A)
Schedule S1: R3(A), R2(A),W3(A), R1(A), W1(A)

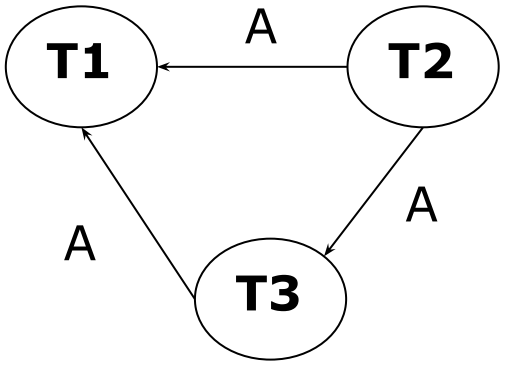


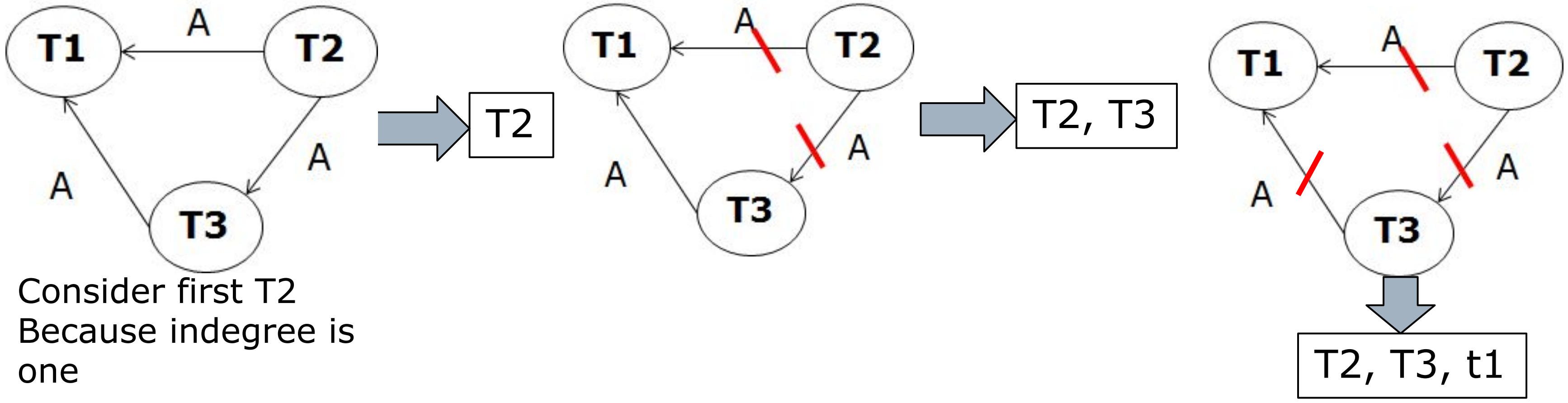
Consider first T2
Because indegree is
one

Solution

- What is the Equivalent serial schedule for non-serial schedule S1:R3(A), R2(A),W3(A), R1(A), W1(A)

Given: Three Transactions
T1 with operations Read(A), Write(A)
T2 with operations Read(A)
T3 with operations Read(A), Write(A)
Schedule S1: R3(A), R2(A),W3(A), R1(A), W1(A)





Consider first T2
Because indegree is
one

Equivalent Serial Schedule is: T2, T3, T1
R2(A), R3(A),W3(A), R1(A), W1(A)

Problem to Solve

Which of the following schedules is (conflict) serializable? For each serializable schedule, determine the equivalent serial schedules.

(a) r1 (X); r3 (X); w1(X); r2(X); w3(X)

(b) r1 (X); r3 (X); w3(X); w1(X); r2(X)

(c) r3 (X); r2 (X); w3(X); r1(X); w1(X)

(d) r3 (X); r2 (X); r1(X); w3(X); w1(X)

Solution

a). *r1(X); r3(X); w1(X); r2(X); w3(X);*
The serialization graph is:

$$\begin{tikzcd}
T1 \arrow[rr] \arrow[dr, shift right] & & T2 \arrow[dl] \\
& T3 \arrow[ul, shift right] &
\end{tikzcd}$$

Not serializable.

b). *r1(X); r3(X); w3(X); w1(X); r2(X);*
The serialization graph is:

$$\begin{tikzcd}
T1 \arrow[rr] \arrow[dr, shift right] & & T2 \\
& T3 \arrow[ul, shift right] \arrow[ur] &
\end{tikzcd}$$

Not serializable.

c). *r3(X); r2(X); w3(X); r1(X); w1(X);*
The serialization graph is:

$$\begin{tikzcd}
T1 & & T2 \arrow[ll] \arrow[dl] \\
& T3 \arrow[ul] &
\end{tikzcd}$$

Serializable.
The equivalent serial schedule is: *r2(X); r3(X); w3(X); r1(X); w1(X);*

d). *r3(X); r2(X); r1(X); w3(X); w1(X);*
The serialization graph is:

$$\begin{tikzcd}
T1 \arrow[dr, shift right] & & T2 \arrow[ll] \arrow[dl] \\
& T3 \arrow[ul, shift right] &
\end{tikzcd}$$

Not serializable.

Problem to Solve

Consider the three transactions T1, T2, and T3, and the schedules S1 and S2 given below. Draw the serializibility (precedence) graphs for S1 and S2 and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).

T1: r1(x); r1(z); w1(x)

T2: r2(z); r2(y); w2(z); w2(y)

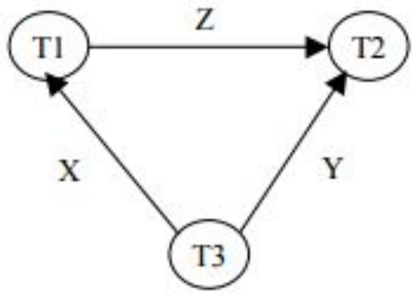
T3: r3(x); r3(y); w3(y)

S1: r1(x); r2(z); r1(z); r3(x); r3(y); w1(x); w3(y); r2(y); w2(z); w2(y)

S2: r1(x); r2(z); r3(x); r1(z); r2(y); r3(y); w1(x); w2(z); w3(y); w2(y)

Solution

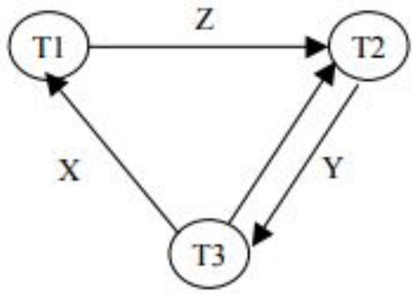
1). The serialization graph for S1 is:


$$\begin{tikzcd} T1 \arrow[rr, "Z"] & & T2 \\ & T3 \arrow[ul, "X"] \arrow[ur, "Y"'] & \end{tikzcd}$$

S1 is serializable. The equivalent serial schedule is: (T3, T1, T2)

r3(X) ; r3(Y); w3(Y); r1(X); r1(Z); w1(X); r2(Z); r2(Y); w2(Z); w2(Y);

2). The serialization graph for S2 is:


$$\begin{tikzcd} T1 \arrow[rr, "Z"] & & T2 \arrow[dl, bend left, "Y"] \\ & T3 \arrow[ul, "X"] \arrow[ur] & \end{tikzcd}$$

S2 is not serializable.

What you have learned until now in Unit4:Transactions

Serial Schedule
Serializable Schedule
Conflict Serializable Schedule (CS)
- Algorithm for testing CS and Converting to Serial Schedule

Characterizing Schedules based on Serializibility

Based on Serializability

1. Conflict Serializable Schedule
2. **View Serializable Schedule**

View Serializability Schedule

- A Schedule is View Serializable if it is **view equivalent** to some **serial schedule**
- What is View equivalence ?

  In **View Equivalence**, respective transactions in the two schedules **read** and **write** the **same data** values

  - Same WR order: **If** in S1: $w_j(A) \rightarrow r_i(A)$
    then in S2: $w_j(A) \rightarrow r_i(A)$
  - First Read: **If** in S1: $r_i(A)$ **reads initial** DB value,
    then in S2: $r_i(A)$ also reads initial DB value
  - Last Write: **If** in S1: $w_i(A)$ does **last write** on A,
    then in S2: $w_i(A)$ also does last write on A

Note: i,j are identifiers of Transactions

View Serializability Schedule

- A Schedule is View Serializable if it is **view equivalent** to some **serial schedule**
- What is View equivalence ?

  In **View Equivalence**, respective transactions in the two schedules **read** and **write** the same **same data** values

  - Same WR order: **If** in S1: $w_j(A)$ –> $r_i(A)$
    then in S2: $w_j(A)$-> $r_i(A)$
  - First Read: **If** in S1: $r_i(A)$ **reads initial** DB value,
    then in S2: $r_i(A)$ also reads initial DB value
  - Last Write: **If** in S1: $w_i(A)$ does **last write** on A,
    then in S2: $w_i(A)$ also does last write on A

Note: i,j are identifiers of Transactions

Note: The premise behind view equivalence:
– "The view": the read operations are said to see *the same view* in both schedules.
– Rule: As long as each read operation of a transaction reads the result of *the same write operation* in both schedules, the write operations of each transaction must produce the same results.

View Equivalence Schedules

View Equivalence, **respective transactions** in the two schedules **read** and **write** the **same data** values

- Initial Reads: **If** in S1: ri(A) **reads initial** DB value,
  then in S2: ri(A) also reads initial DB value
- Same WR order: **If** in S1: wj(A) –> ri(A)
  then in S2: wj(A)-> ri(A)
- Final Writes: **If** in S1: wi(A) does **last write** on A,
  then in S2: wi(A) also does last write on A

Schedule S1

T1	T2	T3
	R(X)	
		R(X)
		W(X)
R(X)		
W(X)		

Schedule S2

T1	T2	T3
		R(X)
	R(X)	
		W(X)
R(X)		
W(X)		

Schedule S2 is view equivalent to serial schedule S1

View Equivalence Schedules

View Equivalence, **respective transactions** in the two schedules **read** and **write** the **same data** values

- Initial Reads: **If** in S1: ri(A) **reads initial** DB value,
  then in S2: ri(A) also reads initial DB value
- Same WR order: **If** in S1: wj(A) –> ri(A)
  then in S2: wj(A)-> ri(A)
- Final Writes: **If** in S1: wi(A) does **last write** on A,
  then in S2: wi(A) also does last write on A

Schedule S2 is view equivalent to serial schedule S1

T1	T2	T3
	R(X)	
		R(X)
		W(X)
R(X)		
W(X)		

Schedule S1

T1	T2	T3
		R(X)
	R(X)	
		W(X)
R(X)		
W(X)		

Schedule S2

Same Initial Reads in S1 and S2

View Equivalence Schedules

View Equivalence, **respective transactions** in the two schedules **read** and **write** the **same data** values

- Initial Reads: **If** in S1: ri(A) **reads initial** DB value,
  then in S2: ri(A) also reads initial DB value
- Same WR order: **If** in S1: wj(A) –> ri(A)
  then in S2: wj(A)-> ri(A)
- Final Writes: **If** in S1: wi(A) does **last write** on A,
  then in S2: wi(A) also does last write on A

Schedule S2 is view equivalent to serial schedule S1

T1	T2	T3
	R(X)	
		R(X)
		W(X)
R(X)		
W(X)		

Schedule S1

T1	T2	T3
		R(X)
	R(X)	
		W(X)
R(X)		
W(X)		

Schedule S2

Same Write Reads orders in S1 and S2

View Equivalence Schedules

View Equivalence, **respective transactions** in the two schedules **read** and **write** the **same data** values

- Initial Reads: **If** in S1: ri(A) **reads initial** DB value,
  then in S2: ri(A) also reads initial DB value
- Same WR order: **If** in S1: wj(A) –> ri(A)
  then in S2: wj(A)-> ri(A)
- Final Writes: **If** in S1: wi(A) does **last write** on A,
  then in S2: wi(A) also does last write on A

Schedule S2 is view equivalent to serial schedule S1

T1	T2	T3
	R(X)	
		R(X)
		W(X)
R(X)		
W(X)		

Schedule S1

T1	T2	T3
		R(X)
	R(X)	
		W(X)
R(X)		
W(X)		

Schedule S2

Same Final Writes in S1 and S2

View Equivalence

View Serializability Summary:

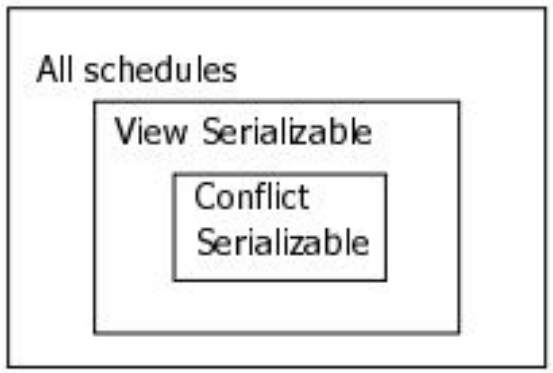
i. Same Transaction **Reads Data First.**

ii. Same **WR Order** of actions.

iii. Same Transaction **Writes Data Last.**

Summarizing Serializability Schedules

- Venn diagram



Note:
Any Conflict serializable schedules is also View serializable schedule but not vice-versa

Note:
- There is an algorithm to test whether a schedule S is view serial schedule.
- However, the problem of testing view serializability has been shown to be NP-hard, meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely.

Characterizing Schedules

Characterizing different schedules based on the following two properties:

A. Based on Serializability

- We shall ignore Commits and Aborts for this section

- Characterize which **schedules are correct** when concurrent transactions are executing.

1. Conflict Serializable Schedule
2. View Serializable Schedule

B. Based on Recoverability

- Commits and Aborts become important for this section

- Characterize which **schedules can be recovered** and how easily.

3. Recoverable Schedule
4. Cascadeless schedule
5. Strict Schedules

Characterizing Schedules based on Recoverability

Based on Recoverability

Characterize which **schedules can be recovered** and how easily.

1. Recoverable Schedule
2. Cascadeless or Avoid Cascading Rollback schedule
3. Strict Schedules

Recoverable Schedule

Recoverability is a situation where we can recover database system to a consistent way after failure.

- Recoverable schedule: A **schedule S** is **recoverable** if no transaction **T** in **S** commits until all transactions **T'**, that have written an item that **T** reads, have committed.

Recoverable Schedule

Recoverability is a situation where we can recover database system to a consistent way after failure.

- Recoverable schedule: A **schedule S** is **recoverable** if no transaction **T** in **S** commits until all transactions **T'**, that have written an item that **T** reads, have committed.

Recoverable Schedule

T1	T2
R(X)	
X=X+10	
W(X)	
Commit	
	R(X)
	X=X-5
	W(X)
	Commit

Note: A committed transaction should never be rolled back

Recoverable Schedule

Recoverability is a situation where we can recover database system to a consistent way after failure.

- Recoverable schedule: A **schedule S** is **recoverable** if no transaction **T** in **S** commits until all transactions **T'**, that have written an item that **T reads, have committed.**

Non-Recoverable Schedule

T1	T2
R(X)	
X=X+10	
W(X)	
	R(X)
	X=X-5
	W(X)
	Commit

Recoverable Schedule

T1	T2
R(X)	
X=X+10	
W(X)	
Commit	
	R(X)
	X=X-5
	W(X)
	Commit

Note: A committed transaction should never be rolled back

Recoverable Schedule

Recoverability is a situation where we can recover database system to a consistent way after failure.

- Recoverable schedule: A **schedule S** is **recoverable** if no transaction **T** in **S** commits until all transactions **T'**, that have written an item that **T** reads, have committed.

Question: Following is Recoverable Schedule ? Yes / No

T1	**T2**
R(X)	
W(X)	
	R(X)
W(X)	
	W(X)
	Commit
Abort	

Recoverable Schedule

Recoverability is a situation where we can recover database system to a consistent way after failure.

- Recoverable schedule: A **schedule S** is **recoverable** if no transaction **T** in **S** commits until all transactions **T'**, that have written an item that **T** reads, have committed.

Question: Following is Recoverable Schedule ? Yes / No

T1	**T2**
R(X)	
W(X)	Dirty Read
	R(X)
W(X)	
	W(X)
	Commit
Abort	

Answer: **NO**

Why NOT recoverable?

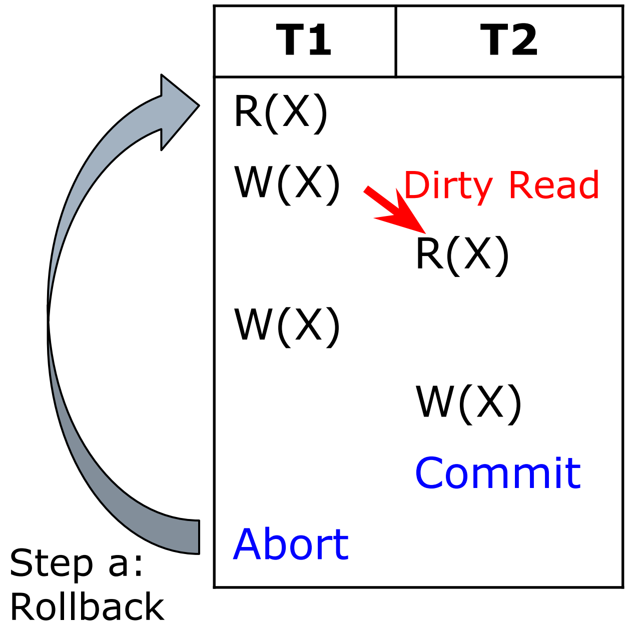
•Because **T2** made a **dirty read** and committed before **T1**

Recoverable Schedule

Recoverability is a situation where we can recover database system to a consistent way after failure.

- Recoverable schedule: A **schedule S** is **recoverable** if no transaction **T** in **S** commits until all transactions **T'**, that have written an item that **T** reads, have committed.

Question: Following is Recoverable Schedule ? Yes / NoRecoverable Schedule



T1	T2
R(X)	
W(X)	
	R(X)
W(X)	
	W(X)
	Commit
Abort	

Answer: NO

Why it not a recoverable schedule?
- Because **T2** made a **dirty read** and committed before **T1**

But why is the schedule Nonrecoverable ?
- Because when the **recovery manager** rolls back (step a) **T1** then A gets its initial value.
- But T2 has already utilized this wrong value and committed something to the DB
- The DB is consequently in an inconsistent state!

Problem to Solve on Recoverable Schedules

- Check whether the following schedule is recoverable schedule

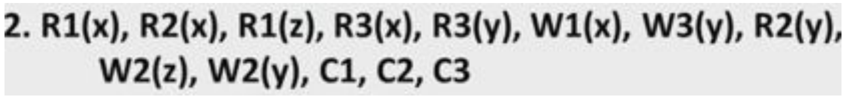
2. R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), W3(y), R2(y),
   W2(z), W2(y), C1, C2, C3

Recoverable schedule: A schedule is recoverable if the following condition is satisfied:
- Tj should commit after Ti if Tj has read any data item written by Ti.

……..Wi(x)……….Rj(x)…………Ci………..Cj -> This schedule is recoverable
……..Wi(x)……….Rj(x)…………Cj………..Ci -> This schedule is non-recoverable

Problem to Solve on Recoverable Schedules

- Check whether the following schedule is recoverable schedule



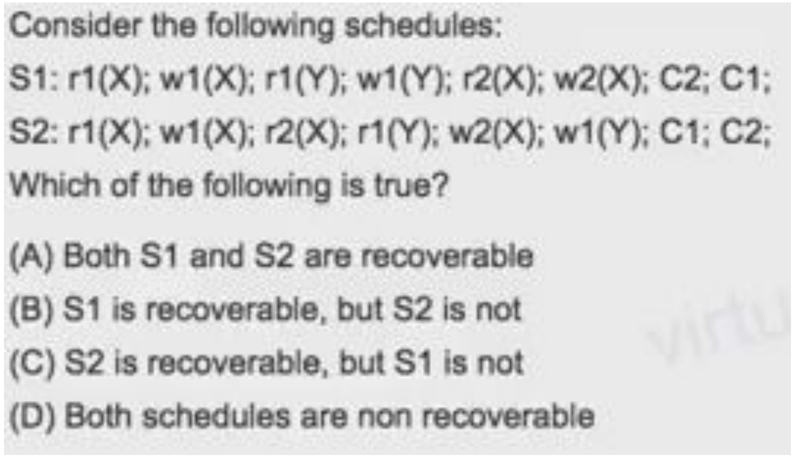
2. R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), W3(y), R2(y),
W2(z), W2(y), C1, C2, C3

Answer: Non-recoverable
Transaction T2 reads the data item R2(y) written by T3 w3(y)
Schedule is non-recoverable because transaction T2 commits C2 before T3 commits C3.

Recoverable schedule: A schedule is recoverable if the following condition is satisfied:
- Tj should commit after Ti if Tj has read any data item written by Ti.

……..Wi(x)……….Rj(x)………..Ci……….Cj -> This schedule is recoverable
……..Wi(x)……….Rj(x)………..Cj……….Ci -> This schedule is non-recoverable

Problem to Solve


Consider the following schedules:

S1: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(X); C2; C1;

S2: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); C1; C2;

Which of the following is true?

(A) Both S1 and S2 are recoverable

(B) S1 is recoverable, but S2 is not

(C) S2 is recoverable, but S1 is not

(D) Both schedules are non recoverable

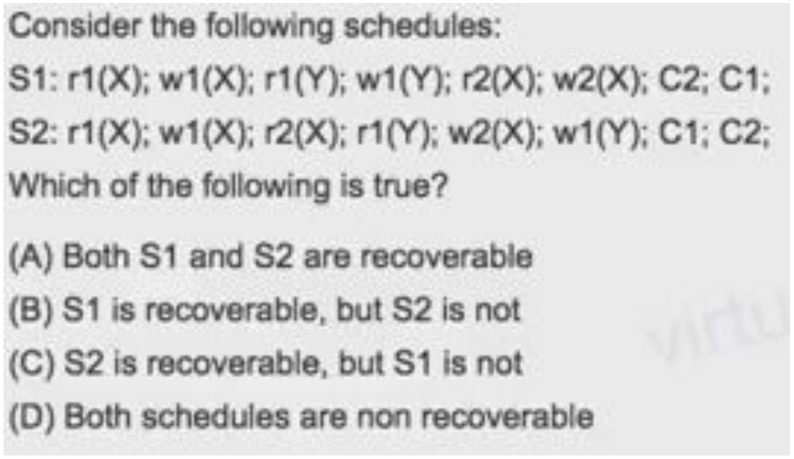
Recoverable schedule: A schedule is recoverable if the following condition is satisfied:

- Tj should commit after Ti if Tj has read any data item written by Ti.

……..Wi(x)……….Rj(x)………..Ci……….Cj -> This schedule is recoverable

……..Wi(x)……….Rj(x)………..Cj……….Ci -> This schedule is non-recoverable

Problem to Solve



Consider the following schedules:

S1: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(X); C2; C1;

S2: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); C1; C2;

Which of the following is true?

(A) Both S1 and S2 are recoverable

(B) S1 is recoverable, but S2 is not

(C) S2 is recoverable, but S1 is not

(D) Both schedules are non recoverable

S1

T1	T2
R1(X)	
W1(X)	
R1(Y)	
W1(Y)	
	R2(X)
	W2(X)
	C2
C1	

S1 Non-Recoverable

S2

T1	T2
R1(X)	
W1(X)	
	R2(X)
R1(Y)	
	W2(X)
W1(Y)	
C1	
	C2

S2 Recoverable

Problem to Solve on Recoverable Schedules

- Check whether the following schedule is recoverable schedule

S4: r1(X); r2(Z); r1(Z); r3(X); r3(Y); w1(X); w3(Y); r2(Y); w2(Z); w2(Y); c1; c2; c3;

Recoverable schedule: A schedule is recoverable if the following condition is satisfied:
- Tj should commit after Ti if Tj has read any data item written by Ti.

……..Wi(x)……….Rj(x)…………Ci………..Cj -> This schedule is recoverable
……..Wi(x)……….Rj(x)…………Cj………..Ci -> This schedule is non-recoverable

Problem to Solve on Recoverable Schedules

- Check whether the following schedule is recoverable schedule

S4: r1(X); r2(Z); r1(Z); r3(X); r3(Y); w1(X); w3(Y); r2(Y); w2(Z); w2(Y); c1; c2; c3;

Answer: Non-recoverable

In S4, T2 reads item Y from T3 but T2 commits before T3 commits. So S4 is **nonrecoverable.**

Recoverable schedule: A schedule is recoverable if the following condition is satisfied:
- Tj should commit after Ti if Tj has read any data item written by Ti.

……..Wi(x)……….Rj(x)………..Ci……….Cj -> This schedule is recoverable
……..Wi(x)……….Rj(x)………..Cj……….Ci -> This schedule is non-recoverable

Characterizing Schedules based on Recoverability

Based on Recoverability

1. Recoverable Schedule
2. **Cascadeless or Avoid Cascading Rollback schedule**
3. Strict Schedules

Cascading Abort

What is Cascading? When effect of one thing is migrated to other and followed by another.

Cascading Abort or
Cascading Rollback

T1	T2	T3
R(X)		
W(X)		
	R(X)	
	W(X)	
		R(X)
		W(X)

Cascading Abort

What is Cascading? When effect of one thing is migrated to other and followed by another.

Cascading Abort or
Cascading Rollback

T1	T2	T3
R(X)		
W(X)		
	R(X)	
	W(X)	
		R(X)
		W(X)
Abort1		
	Abort2	
		Abort3

If system fails here then Transaction T1 aborts similarly T2 aborts and T3 aborts

Cascading Rollback

Cascadeless Schedule

Cascadeless Schedule: Schedule that Avoids Cascading Rollbacks

- A schedule is said to be cascadeless schedule or avoid cascading rollback, if every transaction in the schedule reads only items that were written by committed transactions

Cascadeless Schedule

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

Problem to Solve on Cascadeless Schedules

- Check whether the following schedule is cascadeless schedule

$$S : r_1(x), r_3(y), r_3(x), w_1(x), c_1, w_2(y), r_2(x), w_3(y), c_2, c_3$$

Cascadeless: A schedule is said to be cascadeless, if every transaction in the schedule reads only items that were written by commited (or aborted) transactions.
Cascadeless schedule: A schedule is cascadeless if the following condition is satisfied:
- Tj reads X only ***after*** Ti has written to X and committed (aborted or terminated).

…….Wi(x)………..Ci(x)/Ai(x)………Rj(x) -> This schedule is Cascadeless

Problem to Solve on Cascadeless Schedules

- Check whether the following schedule is cascadeless schedule

$$S : r_1(x), r_3(y), r_3(x), w_1(x), c_1, w_2(y), r_2(x), w_3(y), c_2, c_3$$

Answer: Yes, Cascadeless
T2 reads x after the last transaction that writes x i.e. T1 has committed.
Hence S is Cascadeless

Cascadeless: A schedule is said to be cascadeless, if every transaction in the schedule reads only items that were written by commited (or aborted) transactions.
Cascadeless schedule: A schedule is cascadeless if the following condition is satisfied:
- Tj reads X only ***after*** Ti has written to X and committed (aborted or terminated).

…….Wi(x)………..Ci(x)/Ai(x)……….Rj(x) -> This schedule is Cascadeless

Characterizing Schedules based on Recoverability

Based on Recoverability

1. Recoverable Schedule
2. Cascadeless or Avoid Cascading Rollback schedule
3. **Strict Schedules**

Strict Schedule

- A restrictive type of schedule, called **strict schedule**, in which transaction neither read nor write an item X until the last transaction that wrote X has committed (terminated or aborted).

Formally, if it satisfies the following conditions:

–Tj **reads** a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)

–Tj **writes** a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)

Strict Schedule

T1	**T2**
R(X)	
W(X)	
Commit1	
	R(X)
	W(X)

Problem to Solve on Strict Schedules

- Check whether the following schedule is strict schedule

S3: r1(X); r2(Z); r1(Z); r3(X); r3(Y); w1(X); c1; w3(Y); c3; r2(Y); w2(Z); w2(Y); c2;

Strict schedule: A schedule is strict if it satisfies the following conditions:

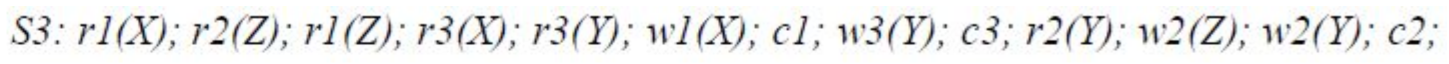
1. Tj reads a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)
2. Tj writes a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)

……..Wi(x)…………Ci(x)………………Rj(x)/Wj(x) -> This schedule is Strict

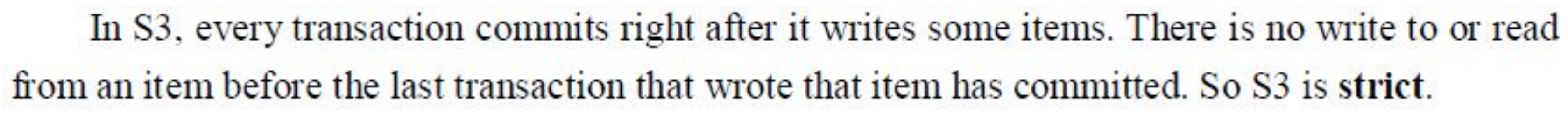
……..Wi(x)…………Rj(x)/Wj(x)…..Ci(x) -> This schedule is not strict

Problem to Solve on Strict Schedules

- Check whether the following schedule is strict schedule


S3: r1(X); r2(Z); r1(Z); r3(X); r3(Y); w1(X); c1; w3(Y); c3; r2(Y); w2(Z); w2(Y); c2;

Answer: Strict


In S3, every transaction commits right after it writes some items. There is no write to or read from an item before the last transaction that wrote that item has committed. So S3 is **strict**.

Strict schedule: A schedule is strict if it satisfies the following conditions:

1. Tj reads a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)
2. Tj writes a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)

……..Wi(x)…………Ci(x)………………Rj(x)/Wj(x) -> This schedule is Strict
……..Wi(x)…………Rj(x)/Wj(x)…..Ci(x) -> This schedule is not strict

Problem to Solve on Strict Schedules

- Check whether the following schedule is strict schedule

S5: r1(X); r2(Z); r3(X); r1(Z); r2(Y); r3(Y); w1(X); c1; w2(Z); w3(Y); w2(Y); c3; c2;

Strict schedule: A schedule is strict if it satisfies the following conditions:

1. Tj reads a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)
2. Tj writes a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)

……..Wi(x)…………Ci(x)………………Rj(x)/Wj(x) -> This schedule is Strict
……..Wi(x)…………Rj(x)/Wj(x)…..Ci(x) -> This schedule is not strict

Problem to Solve on Strict Schedules

- Check whether the following schedule is strict schedule

S5: r1(X); r2(Z); r3(X); r1(Z); r2(Y); r3(Y); w1(X); c1; w2(Z); w3(Y); w2(Y); c3; c2;

Answer: Not, Strict

S5 is **not strict** because T2 writes Y before T3 commits.

Strict schedule: A schedule is strict if it satisfies the following conditions:
1. Tj reads a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)
2. Tj writes a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)

……..Wi(x)…………Ci(x)………………Rj(x)/Wj(x) -> This schedule is Strict
……..Wi(x)…………Rj(x)/Wj(x)…..Ci(x) -> This schedule is not strict

Problem to Solve on Strict Schedules

- Check whether the following schedule is strict schedule

1. R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), C1, W3(y), C3, R2(y), W2(z), W2(y), C2

Strict schedule: A schedule is strict if it satisfies the following conditions:

1. Tj reads a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)
2. Tj writes a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)

……..Wi(x)…………Ci(x)………………Rj(x)/Wj(x) -> This schedule is Strict
……..Wi(x)…………Rj(x)/Wj(x)…..Ci(x) -> This schedule is not strict

Problem to Solve on Strict Schedules

- Check whether the following schedule is strict schedule

1. R1(x), R2(x), R1(z), R3(x), R3(y), W1(x), C1, W3(y), C3, R2(y), W2(z), W2(y), C2

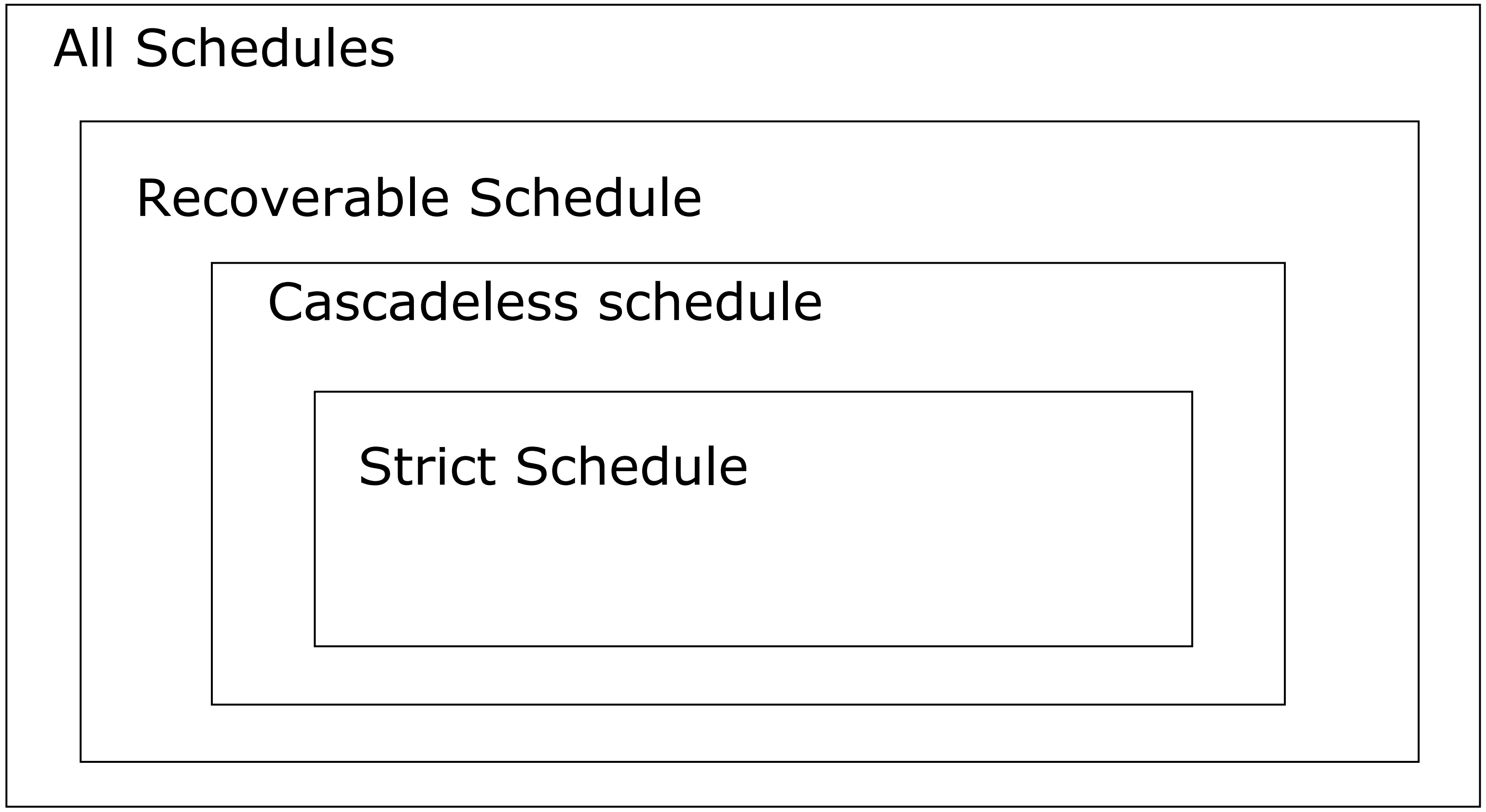
Answer: Yes, Strict

Strict schedule: A schedule is strict if it satisfies the following conditions:

1. Tj reads a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)
2. Tj writes a data item X ***after*** Ti has written to X and Ti is committed (aborted or terminated)

……..Wi(x)…………Ci(x)………………Rj(x)/Wj(x) -> This schedule is Strict
……..Wi(x)…………Rj(x)/Wj(x)…..Ci(x) -> This schedule is not strict

Summary of Schedules based on Recoverability



Note:
- Strict schedules are Cascadeless and Recoverable schedule
- Cascadeless schedules are Recoverable schedule

Transactions and Schedule

- Transactions and Schedules

– Serial Schedule … one after the other…

– Complete Schedule … with Commit, Abort

Serializability

– "Correctness Measure" of some Schedule

– Why is it useful? It answers the question: "Will an interleaved schedule execute correctly

– i.e., a Serializable schedule will execute as correctly as serial schedule … but in an interleaved manner!

Recoverability

– "Recoverability Measure" of some Schedule.

– Why is it useful? It answers the question: "Do we need to rollback a some (or all) transactions in an interleaved schedule after some Failure (e.g., ABORT)"

– i.e., in a Recoverable schedule no transaction needs to be rolled back once committed!

Properties of Transactions

ACID Properties

1. **A**tomicity
2. **C**onsistency preservation
3. **I**solation
4. **D**urability or permanency

Properties of Transactions

1.Atomicity

- Either all operations of a transaction occurs or none
- Their should not be the case where half of the operations

of transaction has been executed and other remaining half of the operations has not been executed.

Example of Transaction:

Increasing salary of an employee by 10%.

The three operations of this transactions are:

read(salary),salary=salary+salary*0.10,write(salary)

All this three operations should be executed for the transaction to be successful

Properties of Transactions

2. **C**onsistency preservation (or Correctness)

A transaction should lead database from one consistent state to another consistent state.

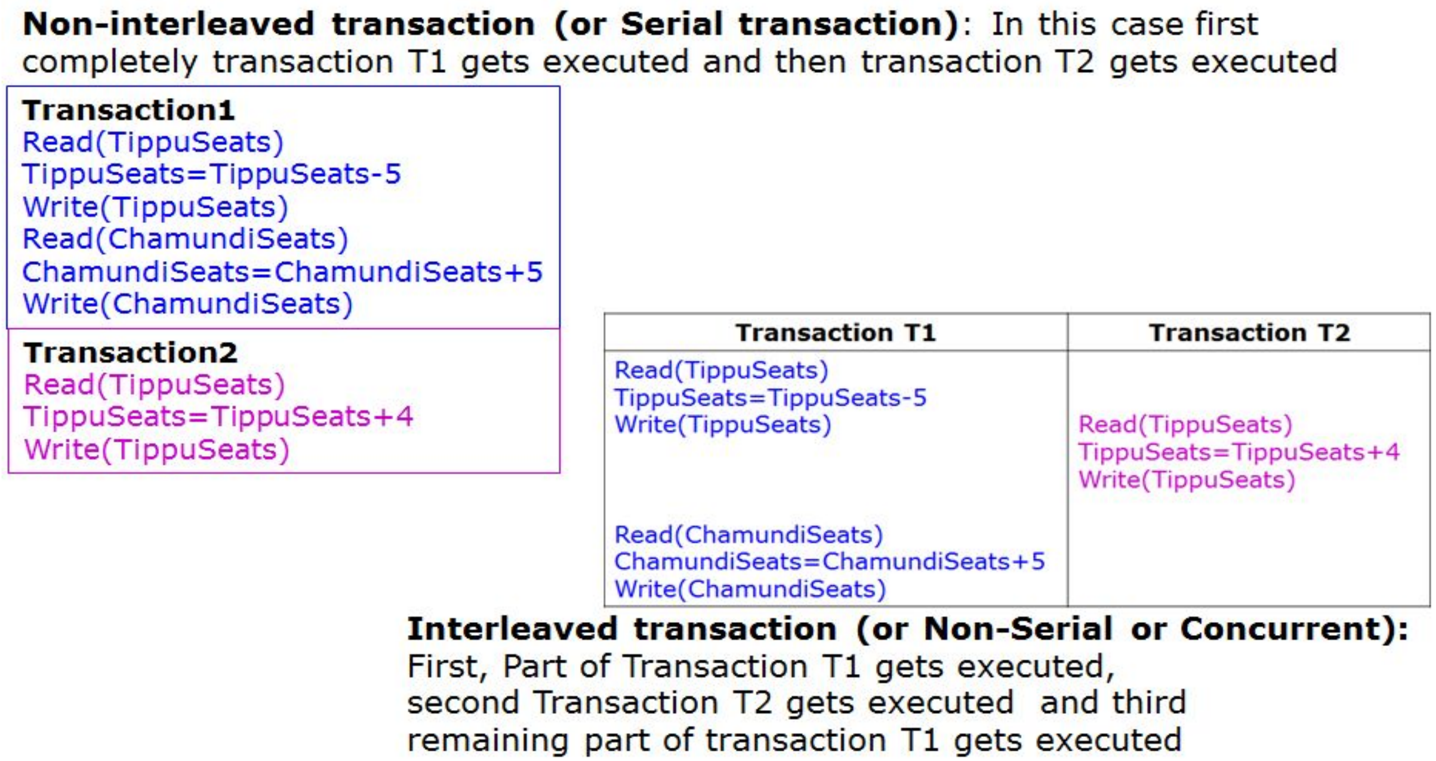
Example: Say in a database table if both Date of Birth (**DOB**) and **Age** values are stored.

If any transaction changes DOB then appropriately the change in age value should reflected in database table. When DOB is changed but Age value has not been changed then table data will not be in consistent state.

Properties of Transactions

3. **I**solation

Ensures that concurrent execution results in a system state that would be obtained if transaction would be executed serially.



Non-interleaved transaction (or Serial transaction): In this case first completely transaction T1 gets executed and then transaction T2 gets executed

Transaction1
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)

Transaction2
Read(TippuSeats)
TippuSeats=TippuSeats+4
Write(TippuSeats)

Transaction T1	Transaction T2

Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)	

	Read(TippuSeats)
	TippuSeats=TippuSeats+4
	Write(TippuSeats)

Read(ChamundiSeats)
ChamundiSeats=ChamundiSeats+5
Write(ChamundiSeats)	

Interleaved transaction (or Non-Serial or Concurrent): First, Part of Transaction T1 gets executed, second Transaction T2 gets executed and third remaining part of transaction T1 gets executed

Properties of Transactions

4. **D**urability or permanency

Changes should be permanent. The changes must NOT be lost due to some database failure.

Summarizing Properties of Transactions

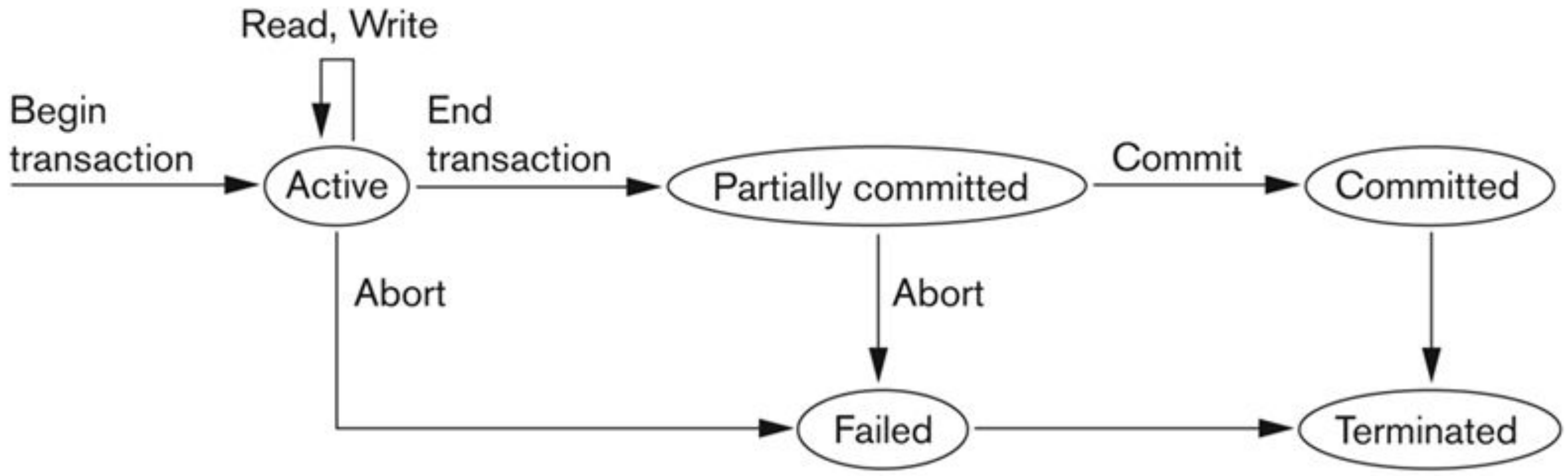
ACID Properties

1. **A**tomicity
2. **C**onsistency preservation
3. **I**solation
4. **D**urability or permanency

Transaction

- A transaction is an atomic unit of work that is either completed in its entirety or not done at all.
  - For **recovery purposes**, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states**:
  - Active state
  - Partially committed state
  - Committed state
  - Failed state
  - Terminated State

State transition diagram illustrating the states for transaction execution



Active, the initial state; the transaction stays in this state while it is executing
Partially committed, after the final statement has been executed.
Failed, after the discovery that normal execution can no longer proceed.
Aborted, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:

  restart the transaction – only if no internal logical error
  kill the transaction

Committed, after successful completion

Next what you are going learn is………

Concurrency Control Protocols

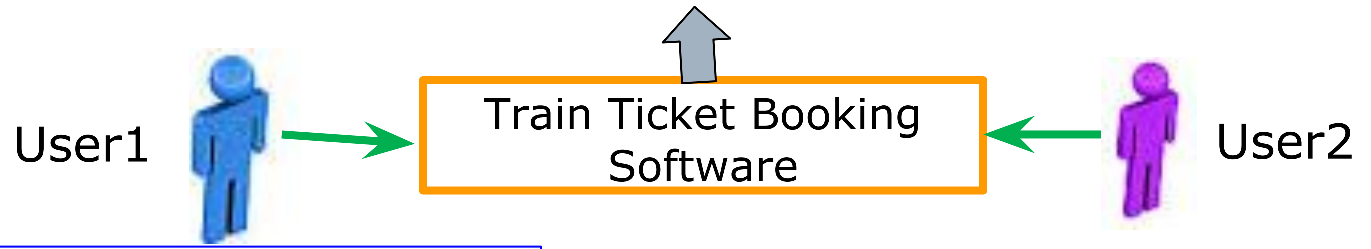
- To ensure when to give access to data item when transactions are getting executed concurrently or in interleaved way

Why we need Concurrency Control Protocols ?

Consider an example of Train Reservation System

Train Reservation Database

Source	Destination	Train Name	Start Time	Start Date	Availability of Seats
Bangalore	Mysore	Tippu Express	15:00	25-3-2016	80
Bangalore	Mysore	Chamundi Express	18:15	25-3-2016	70



User1: Wants to **reserve** 5 seats on **Tippu** express

Transaction **T1**

```
Read(TippuSeats)
TippuSeats=TippuSeats-5
Write(TippuSeats)
```

User2: Wants to **reserve** 4 seats on **Tippu** express

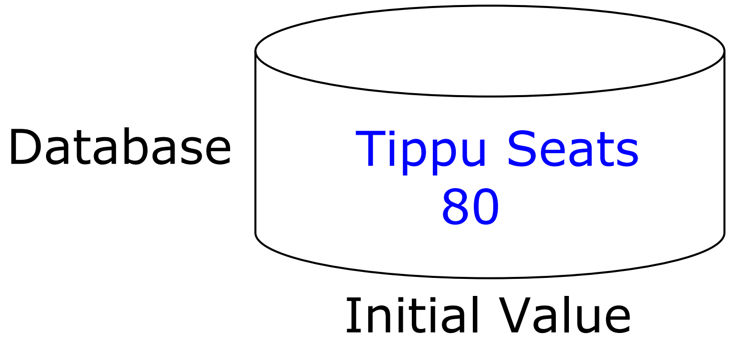
Transaction **T2**

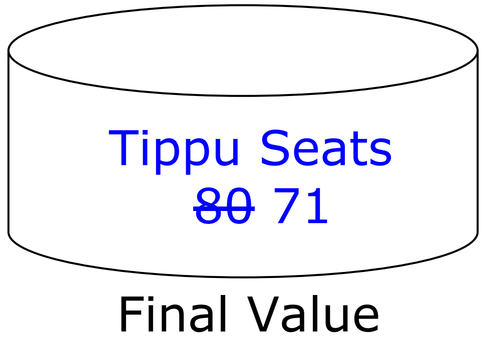
```
Read(TippuSeats)
TippuSeats=TippuSeats-4
Write(TippuSeats)
```

Why we need Concurrency Control Protocols ?

Case 1: When the **User1** Logins at **4:00pm** to booking system and **User 2** Logins at **4:03pm** to booking system

Time	**T1**	**T2**
4:00pm	Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	
4:02pm	Write(TippuSeats)	
4:03pm		Read(TippuSeats)
4:04pm		TippuSeats=TippuSeats-4
4:05pm		Write(TippuSeats)

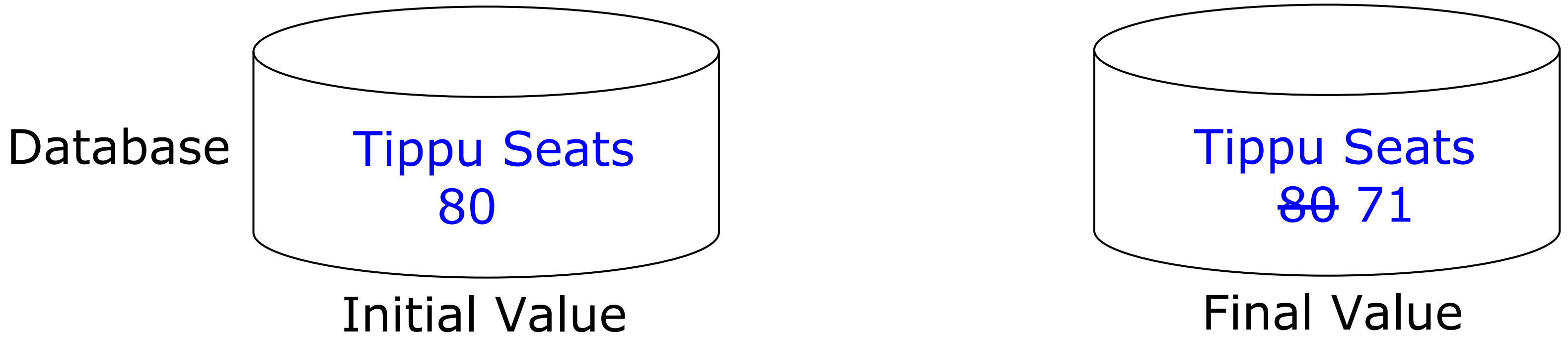




Why we need Concurrency Control Protocols ?

Case 2: When the **User1** Logins at **4:03pm** to booking system and **User 2** Logins at **4:00pm** to booking system

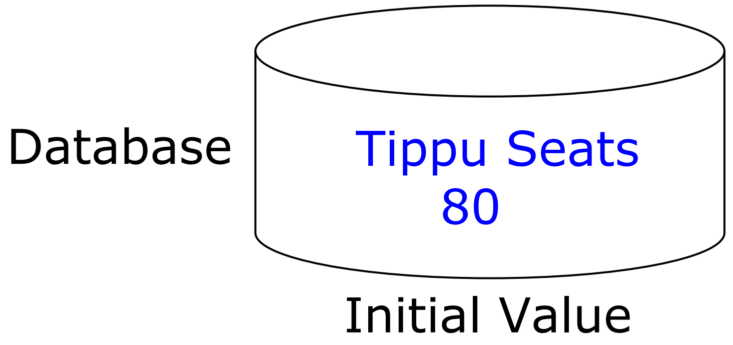
Time	T1	T2
4:00pm		Read(TippuSeats)
4:01pm		TippuSeats=TippuSeats-4
4:02pm		Write(TippuSeats)
4:03pm	Read(TippuSeats)	
4:04pm	TippuSeats=TippuSeats-5	
4:05pm	Write(TippuSeats)	



Why we need Concurrency Control Protocols ?

Case 3: When the **User1** Logins at **4:00**pm to booking system and **User 2** Logins at **4:01**pm to booking system

Time	T1	T2
4:00pm	Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	Read(TippuSeats)
4:02pm	Write(TippuSeats)	TippuSeats=TippuSeats-4
4:03pm		Write(TippuSeats)
4:04pm		
4:05pm		

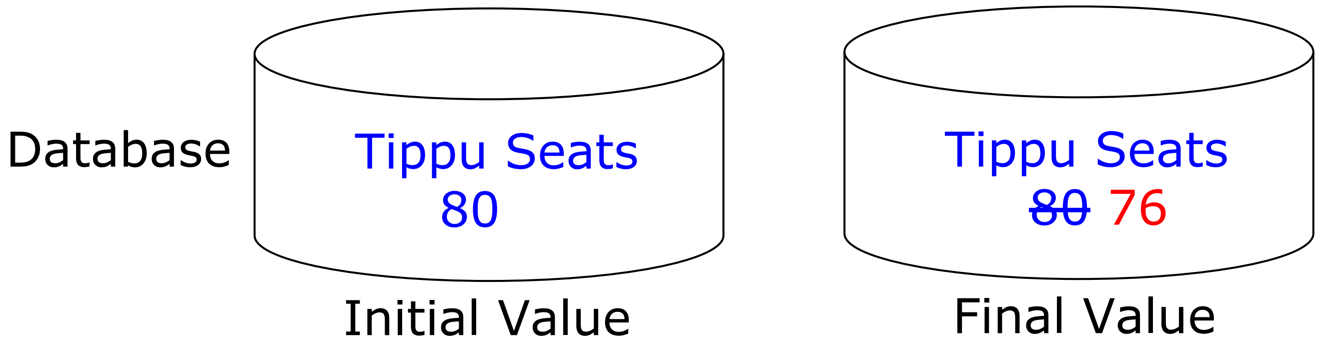


Question:
What will be the final value Of TippuSeats when above Transactions are executed ?

Why we need Concurrency Control Protocols ?

Case 3: When the User1 Logins at 4:00pm to booking system and User 2 Logins at 4:01pm to booking system

Time	T1	T2
4:00pm	Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	Read(TippuSeats)
4:02pm	Write(TippuSeats)	TippuSeats=TippuSeats-4
4:03pm		Write(TippuSeats)
4:04pm		
4:05pm		

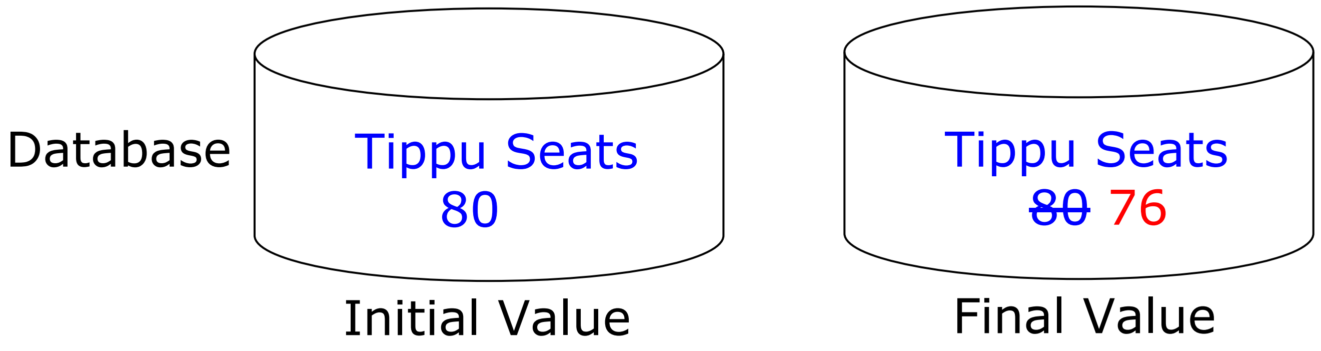


Database is in Inconsistent state WRONG value **76** Updaed for Tippu Seats

Why we need Concurrency Control Protocols ?

Case 3: When the User1 Logins at 4:00pm to booking system and User 2 Logins at 4:01pm to booking system

Time	T1	T2
4:00pm	Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	Read(TippuSeats)
4:02pm	Write(TippuSeats)	TippuSeats=TippuSeats-4
4:03pm		Write(TippuSeats)
4:04pm		
4:05pm		



Database is in Inconsistent state WRONG value **76** Updaed for Tippu Seats

To avoid this type of Problems Concurrency Control Protocols will be used

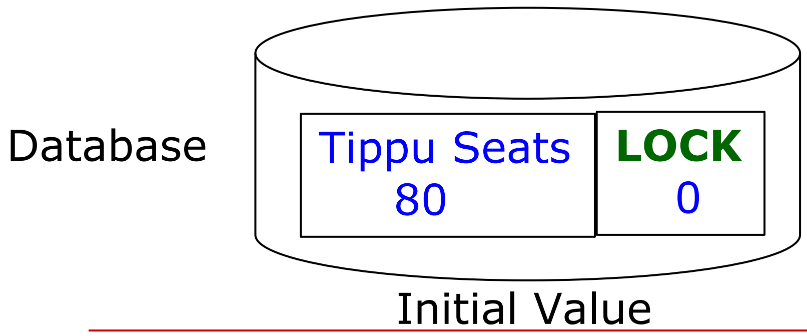
Why we need Concurrency Control Protocols ?

- When two or more users or transactions wants to access the **same data item** then concurrency control protocols should be followed to leave database to a consistent state after completion of executing transactions.
- One type Concurrency protocols are **LOCK based protocols**
- Under LOCK based protocol any transaction that needs to access the data item should first obtain the LOCK on the data item

Concurrency Control Protocol: LOCK based

For the following transactions schedule we will see by using LOCK based protocol, how to obtain correct value i.e., leaving the Database system in a consistent state after execution of transactions

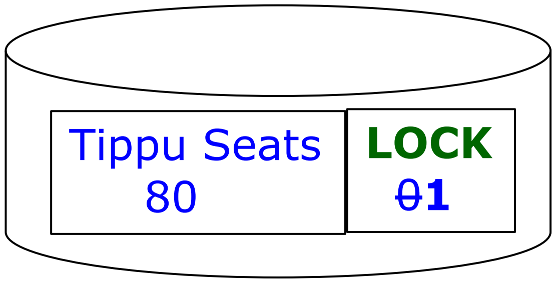
Time	T1	T2
4:00pm	Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	Read(TippuSeats)
4:02pm	Write(TippuSeats)	TippuSeats=TippuSeats-4
4:03pm		Write(TippuSeats)
4:04pm		
4:05pm		



Concurrency Control Protocol: LOCK based

Time	T1	T2
4:00pm	**LOCK**(TippuSeats), Read(TippuSeats)	
4:01pm		
4:02pm		
4:03pm		
4:04pm		
4:05pm		

Database

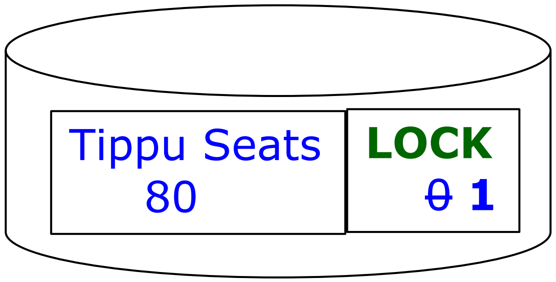


When Transaction T1 executes, LOCK(TippuSeats), LOCK variable value of TippuSeats will be changed from zero to one

Concurrency Control Protocol: LOCK based

Time	T1	T2
4:00pm	**LOCK**(TippuSeats), Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	T2 will **WAIT** because it cannot obtain LOCK on data item TippuSeats
4:02pm		
4:03pm		
4:04pm		
4:05pm		

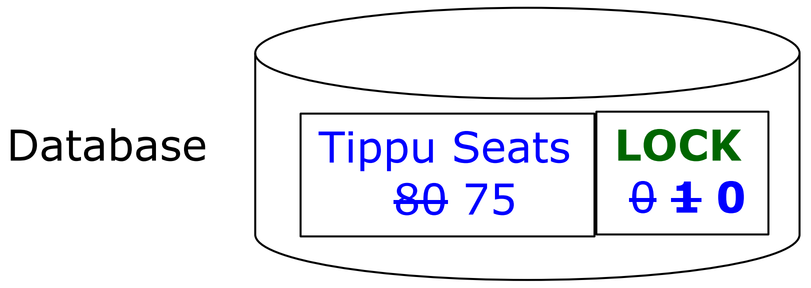
Database



When transaction T2 comes at 4:01pm,
Then T2 will check for LOCK variable value
Of TippuSeats it is **ONE** so T2 **WAITS**

Concurrency Control Protocol: LOCK based

Time	T1	T2
4:00pm	**LOCK**(TippuSeats), Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	T2 will **WAIT** because it cannot obtain LOCK on data item TippuSeats
4:02pm	Write(TippuSeats), **UNLOCK**(TippuSeats)	**WAIT**
4:03pm		
4:04pm		
4:05pm		

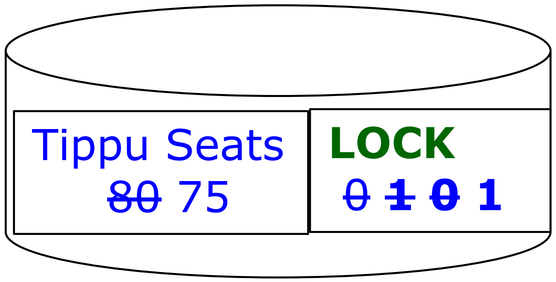


When transaction T1 executes UNLOCK(TippuSeats), LOCK value of TippuSeats will be changed from ONE to ZERO

Concurrency Control Protocol: LOCK based

Time	T1	T2
4:00pm	**LOCK**(TippuSeats), Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	T2 will **WAIT** because it cannot obtain LOCK on data item TippuSeats
4:02pm	Write(TippuSeats), **UNLOCK**(TippuSeats)	**WAIT**
4:03pm		**LOCK**(TippuSeats), Read(TippuSeats)
4:04pm		
4:05pm		

Database

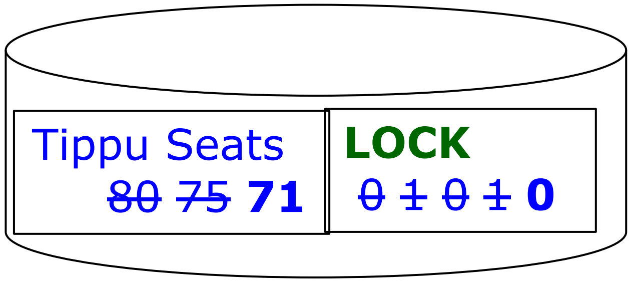


When transaction T2 executes LOCK(TippuSeats), LOCK value of TippuSeats will be changed from ZERO to ONE

Concurrency Control Protocol: LOCK based

Time	T1	T2
4:00pm	**LOCK**(TippuSeats), Read(TippuSeats)	
4:01pm	TippuSeats=TippuSeats-5	T2 will **WAIT** because it cannot obtain LOCK on data item TippuSeats
4:02pm	Write(TippuSeats), **UNLOCK**(TippuSeats)	**WAIT**
4:03pm		**LOCK**(TippuSeats), Read(TippuSeats)
4:04pm		TippuSeats=TippuSeats-4
4:05pm		Write(TippuSeats), **UNLOCK**(TippuSeats)

Database



Now final value of TippuSeats is 71, which is CORRECT

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data. LOCKS are used to ensure concurrency.
- Two types of locks:
  - ***Shared*** (S) locks (also called ***read*** *locks)*
    - Obtained if transactions want to only read an item
  - ***Exclusive*** (X) locks (also called ***write*** *locks)*
    - Obtained for updating a data item

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data. LOCKS are used to ensure concurrency.
- Two types of locks:
  - ***Shared*** (S) locks (also called ***read*** *locks)*
    - Obtained if transactions want to only read an item
  - ***Exclusive*** (X) locks (also called ***write*** *locks)*
    - Obtained for updating a data item

		User 1, HOLDER of the LOCK on data item X	
		Shared or Read Lock	Write or Exclusive Lock
User2, REQUESTOR of LOCK for data item X	Shared or Read Lock		
	Write or Exclusive Lock		

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
  - ***Shared*** (S) locks (also called ***read*** *locks)*
    - Obtained if we want to only read an item
  - ***Exclusive*** (X) locks (also called ***write*** *locks)*
    - Obtained for updating a data item

		User 1, HOLDER of the LOCK on data item X	
		Shared or Read Lock	Write or Exclusive Lock
User2, REQUESTOR of LOCK for data item X	Shared or Read Lock	**YES**: USER 2 will granted with Read lock	
	Write or Exclusive Lock		

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
  - ***Shared*** (S) locks (also called ***read*** *locks)*
    - Obtained if transactions want to only read an item
  - ***Exclusive*** (X) locks (also called ***write*** *locks)*
    - Obtained for updating a data item

		User 1, HOLDER of the LOCK on data item X	
		Shared or Read Lock	Write or Exclusive Lock
User2, REQUESTOR of LOCK for data item X	Shared or Read Lock	**YES**: USER 2 will granted with Read lock	**NO**: USER 2 will not be granted with Read lock
	Write or Exclusive Lock		

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
  - ***Shared*** (S) locks (also called ***read*** *locks)*
    - Obtained if transactions want to only read an item
  - ***Exclusive*** (X) locks (also called ***write*** *locks)*
    - Obtained for updating a data item

		User 1, HOLDER of the LOCK on data item X	
		Shared or Read Lock	Write or Exclusive Lock
User2, REQUESTOR of LOCK for data item X	Shared or Read Lock	**YES**: USER 2 will granted with Read lock	**NO**: USER 2 will not be granted with Read lock
	Write or Exclusive Lock	**??**	**??**

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
  - ***Shared*** (S) locks (also called ***read*** *locks)*
    - Obtained if transactions want to only read an item
  - ***Exclusive*** (X) locks (also called ***write*** *locks)*
    - Obtained for updating a data item

		User 1, HOLDER of the LOCK on data item X	
		Shared or Read Lock	Write or Exclusive Lock
User2, REQUESTOR of LOCK for data item X	Shared or Read Lock	**YES**: USER 2 will granted with Read lock	**NO**: USER 2 will not be granted with Read lock
	Write or Exclusive Lock	**NO**: USER 2 will not be granted with Write lock	**NO**: USER 2 will not be granted with Write lock

Lock instructions

- New LOCK instructions
  - Lock-S: shared (or read) lock request
  - Lock-X: exclusive (or write) lock request
  - Unlock: release previously held lock
- Example schedule:

T1

```
read(B)
B =B-50
write(B)
read(A)
A =A + 50
write(A)
```

T2

```
read(A)
read(B)
display(A+B)
```

Lock instructions

- New LOCK instructions
  - Lock-S: shared (or read) lock request
  - Lock-X: exclusive (or write) lock request
  - Unlock: release previously held lock
- Example schedule:

T1	T2
Lock-X(B)	Lock-S(A)
read(B)	read(A)
B = B-50	Unlock(A)
write(B)	Lock-S(B)
Unlock(B)	read(B)
Lock-X(A)	Unlock(B)
read(A)	display(A+B)
A =A + 50	
write(A)	
Unlock(A)	

Lock-based Protocols

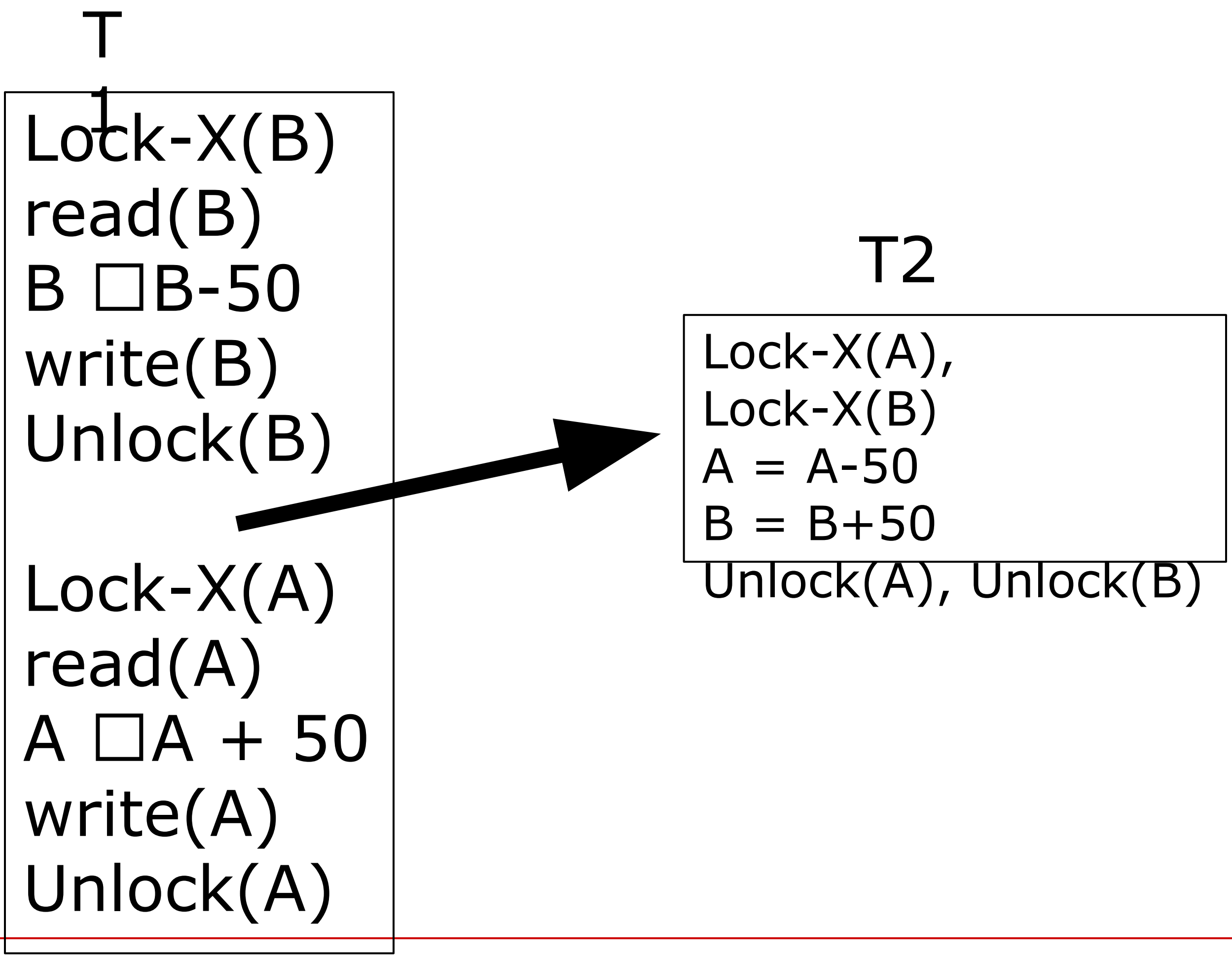
- Lock requests are made to the *concurrency control manager*
  - It decides whether to *grant* a lock request
- T1 asks for a lock on data item A, and T2 currently has a lock on it ?
  - Depends

T2 lock type	**T1 lock type**	**Should allow ?**
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

  - If *compatible,* grant the lock, otherwise T1 waits in a *queue.*

Lock-based Protocols

- How do we actually use this to guarantee serializability/recoverability ?
  - Not enough just to take locks when you need to read/write something



T
1

```
Lock-X(B)
read(B)
B □B-50
write(B)
Unlock(B)

Lock-X(A)
read(A)
A □A + 50
write(A)
Unlock(A)
```

T2

```
Lock-X(A),
Lock-X(B)
A = A-50
B = B+50
Unlock(A), Unlock(B)
```

Two-Phase Locking Protocol (2PL)

A transaction is said to follow two phase locking protocol if all locking operations (read-lock or shared lock, write-lock or Exclusive- lock) precede the first Unlock operation in the transaction. Such transaction can be divided into two phases

- Phase 1: Growing (or Lock Acquiring) phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking (or Lock Releasing) phase
  - Transaction may only release locks

Two-Phase Locking Protocol (2PL)

- Phase 1: Growing (or Lock Acquiring) phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking (or Lock Releasing) phase
  - Transaction may only release locks

T1: Not following 2PL

```
Lock-X(B)
read(B)
B □B-50
write(B)
Unlock(B)

Lock-X(A)
read(A)
A □A + 50
write(A)
Unlock(A)
```

Two-Phase Locking Protocol (2PL)

- Phase 1: Growing (or Lock Acquiring) phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking (or Lock Releasing) phase
  - Transaction may only release locks

T1: Not following 2PL

```
Lock-X(B)
read(B)
B ☐B-50
write(B)
Unlock(B)

Lock-X(A)
read(A)
A ☐A + 50
write(A)
Unlock(A)
```

Two-Phase Locking Protocol (2PL)

- Phase 1: Growing (or Lock Acquiring) phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking (or Lock Releasing) phase
  - Transaction may only release locks

T1: Not following 2PL

```
Lock-X(B)
read(B)
B = B-50
write(B)
Unlock(B)

Lock-X(A)
read(A)
A =A + 50
write(A)
Unlock(A)
```

T1: Follows 2PL

Growing phase:

```
Lock-X(B)
read(B)
B = B - 50
write(B)
Lock-X(A)
read(A)
A = A - 50
write(A)
```

Shrinking phase:

```
unlock(B)
unlock(A)
```

Variations of 2PL

1. Basic 2PL

- Two Phase: Growing and Shrinking Phase

2. Conservative (or static) 2PL

- Conservative 2PL requires a transaction to lock all the items it accesses before the transaction begins execution by pre declaring its write set and read set.
- If any of the pre declared items needed cannot be locked, the transaction does not lock any item; instead it **waits** until all the items are available for locking

3. Strict 2PL

- A transaction T **does not release any of its exclusive (write)** locks until after it **commits** or aborts.
- Hence no other transaction can read or write an item that is written by T unless T has committed.

4. Rigorous 2PL

- A transaction T **does not release any of its locks (exclusive or shared)** until after it **commits** or aborts and so it is easier to implement than strict 2PL.

Database Recovery Systems

Database Recovery Systems looks to achieve
- Transaction Atomicity
- Transaction Durability

Why recovery is Needed ?

To avoid following failures (or What causes a Transaction to fail)

1. **A computer failure (system crash):** A **hardware or software error** occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.
2. **A transaction or system error:** Some operation in the transaction may cause it to fail, such as **integer overflow or division by zero**. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
3. Local errors or **exception conditions detected by the transaction**: Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as **insufficient account balance in a banking database**, may cause a transaction, such as a fund withdrawal from that account, to be canceled. A programmed abort in the transaction causes it to fail.
4. **Concurrency control enforcement**: The concurrency control method may decide to abort the transaction, to be restarted later, because it **violates serializability** or because several transactions are in a **state of deadlock**.
5. **Disk failure**: Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. **Physical problems and catastrophes**: This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks by mistake.

Failure Classification

Transaction failure :

- Logical errors: transaction cannot complete due to some internal error condition
- System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

System crash: a power failure or other hardware or software failure causes the system to crash.

- Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted by system crash
  - Database systems have numerous integrity checks to prevent corruption of disk data

Disk failure: a head crash or similar disk failure destroys all or part of disk storage

- Destruction is assumed to be detectable: disk drives use checksums to detect failures

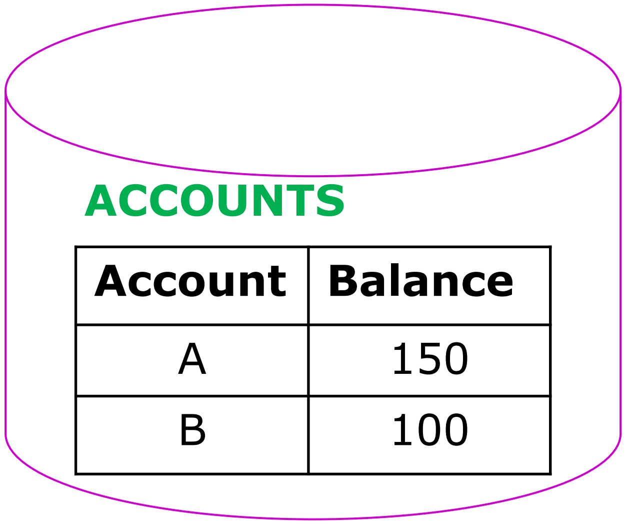
Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts

  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Example: Database Transaction

- Transferring Rs.100/- from account A to account B


ACCOUNTS

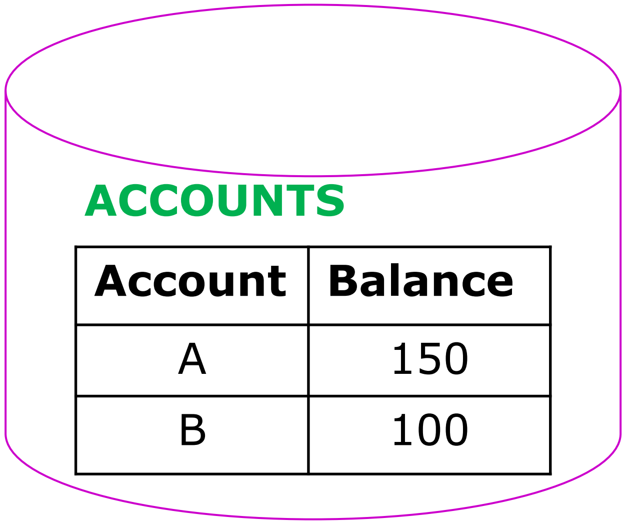
Account	Balance
A	150
B	100

Example: Database Transaction

- Transferring Rs.100/- from account A to account B

Transaction T1

Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)



ACCOUNTS

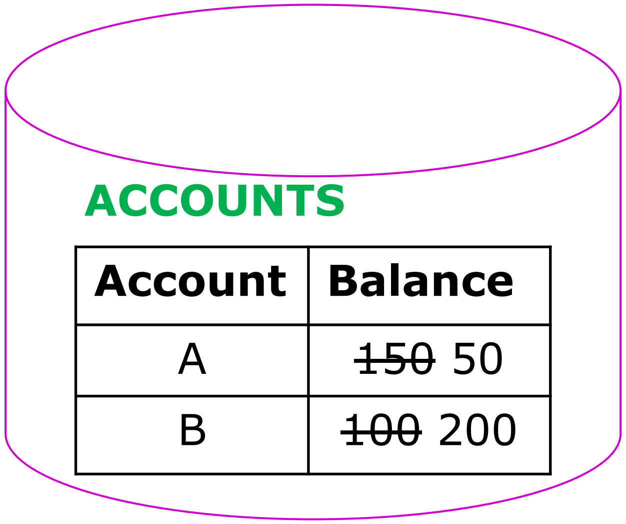
Account	Balance
A	150
B	100

Example: Database Transaction

- Transferring Rs.100/- from account A to account B

Transaction T1

Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)

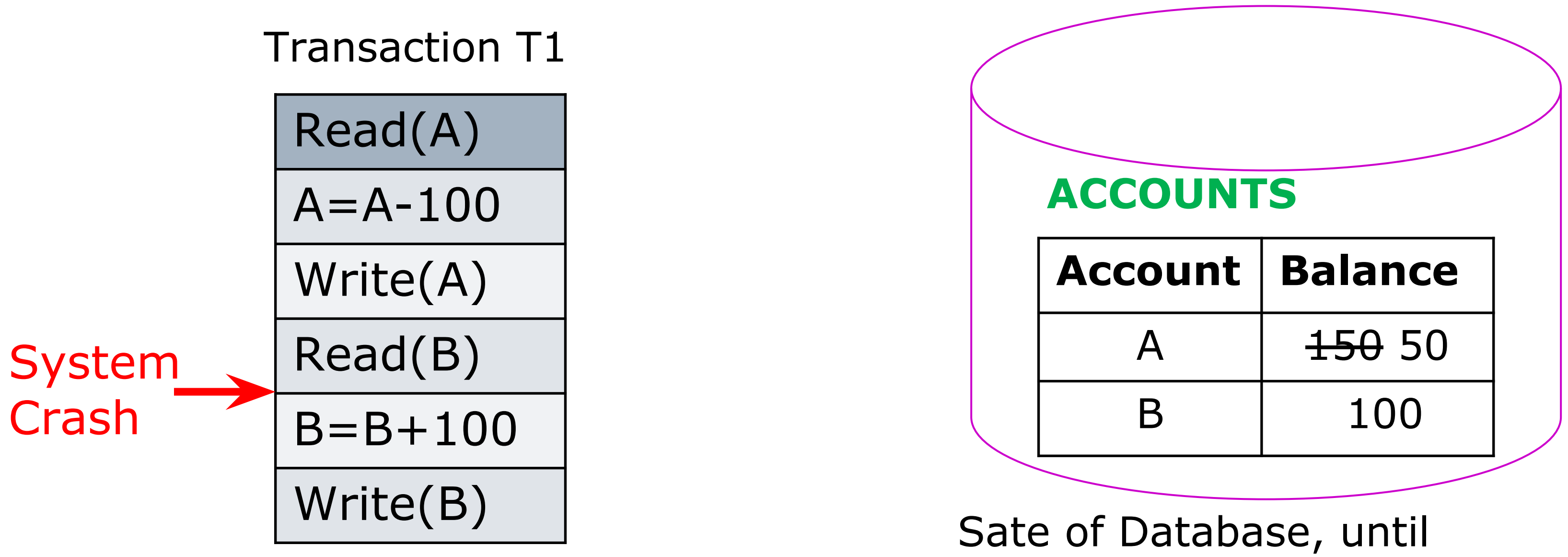


Account	Balance
A	~~150~~ 50
B	~~100~~ 200

Sate of Database, After completely executing all operations of Transaction T1

Example: Database Transaction

- Transferring Rs.100/- from account A to account B

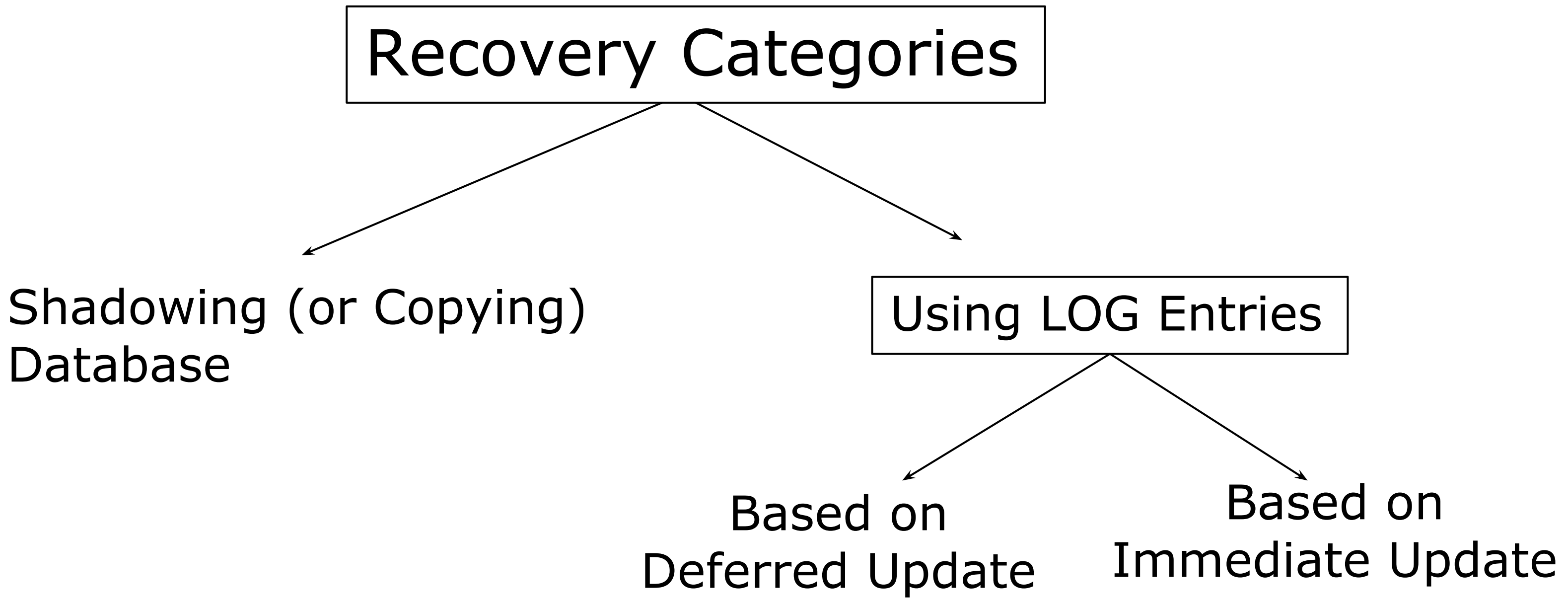


Account	Balance
A	~~150~~ 50
B	100

Sate of Database, until
The system crash.

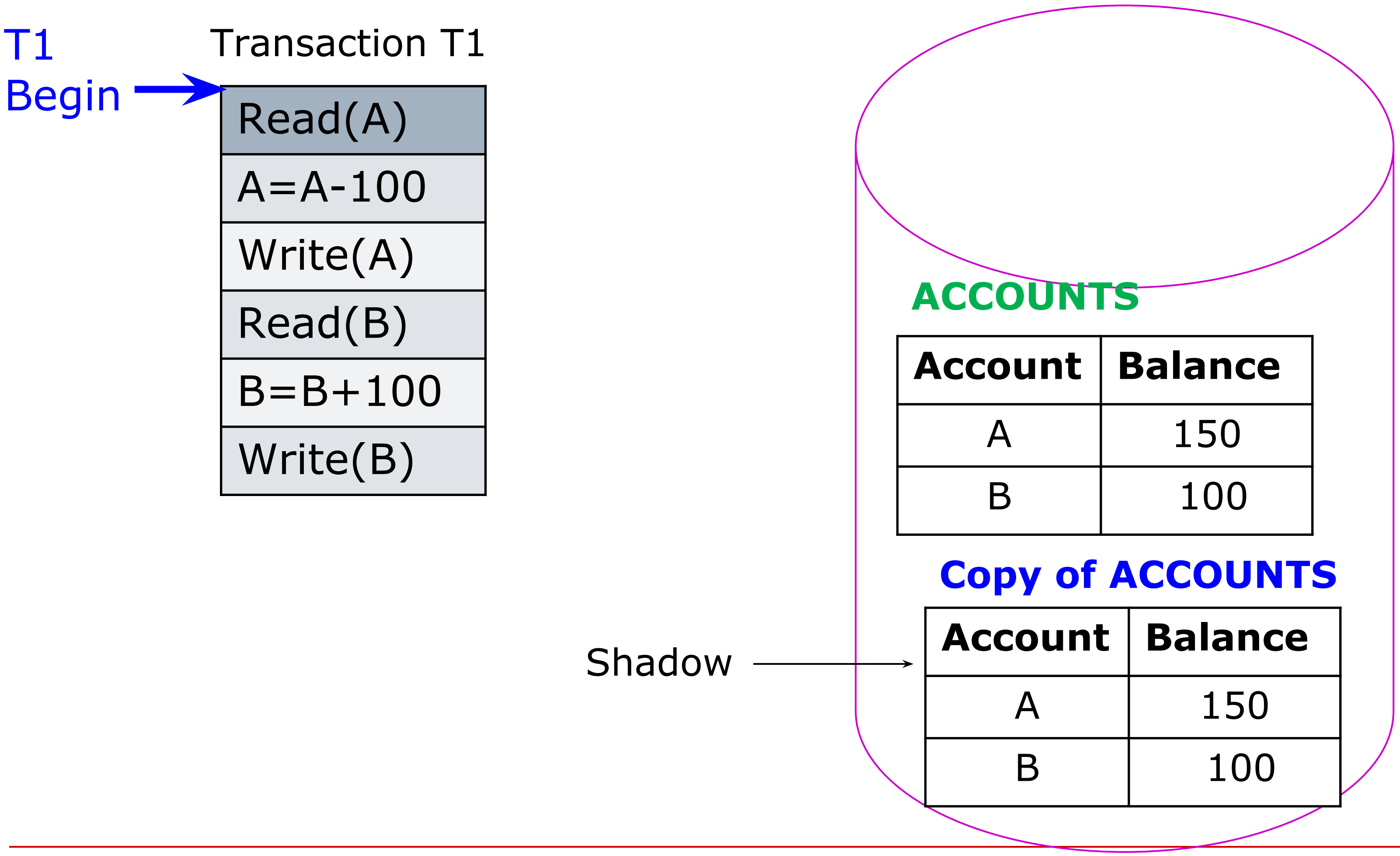
Now database is inconsistent

Database Recovery Mechanisms



Recovery using Shadowing

- Transferring Rs.100/- from account A to account B



ACCOUNTS

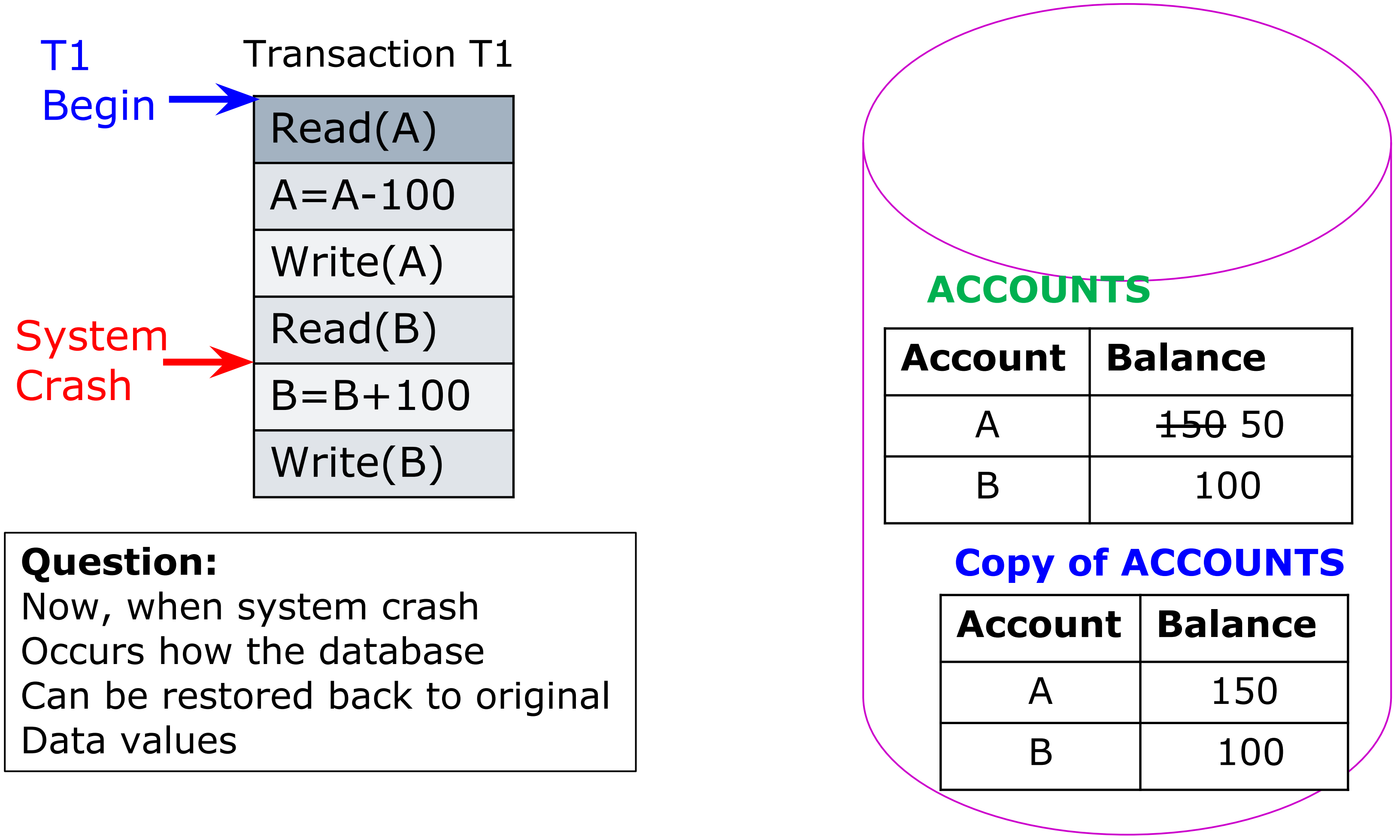
Account	Balance
A	150
B	100

Copy of ACCOUNTS

Account	Balance
A	150
B	100

Recovery using Shadowing

- Transferring Rs.100/- from account A to account B



Transaction T1
Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)

ACCOUNTS

Account	Balance
A	~~150~~ 50
B	100

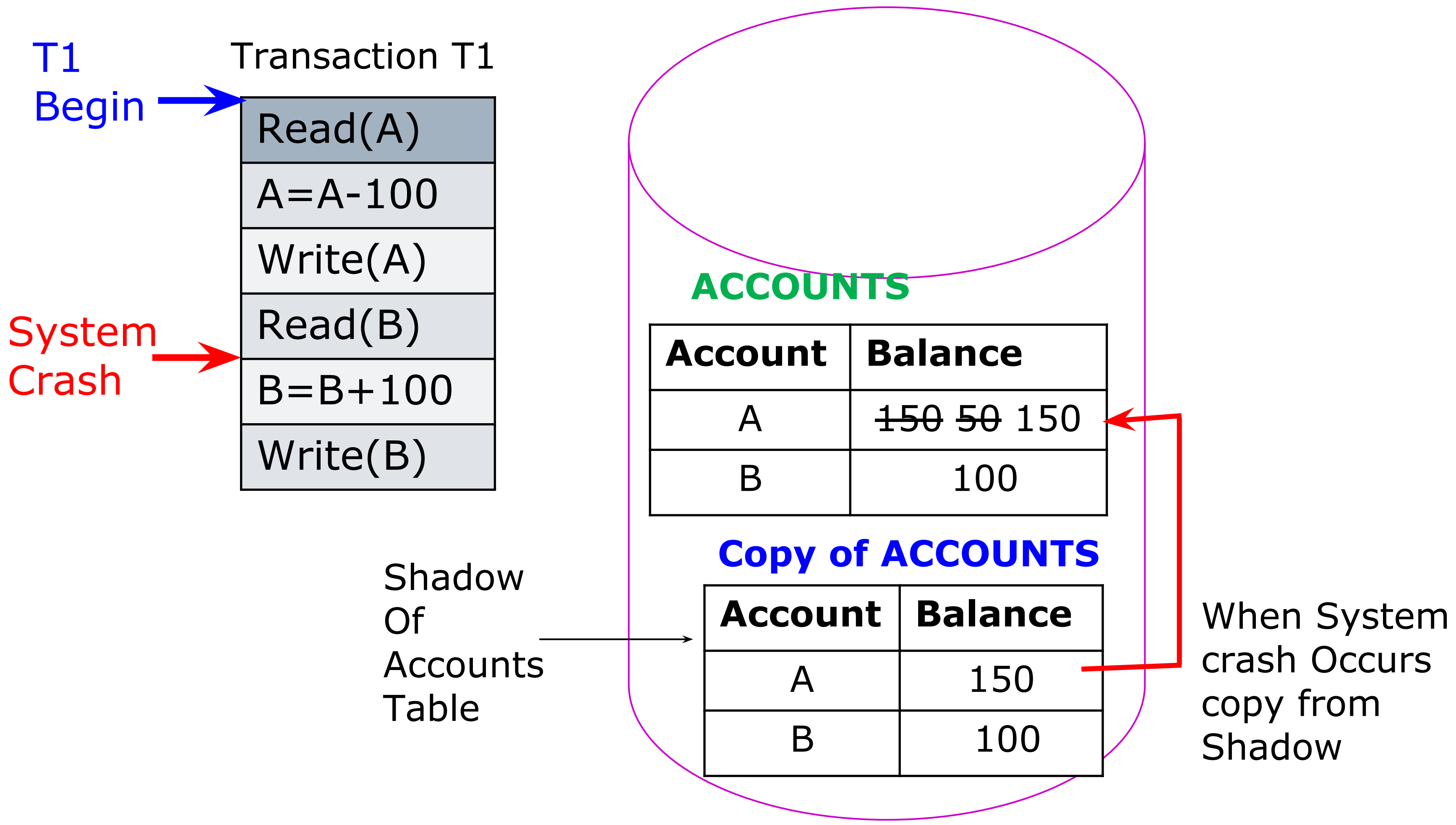
Copy of ACCOUNTS

Account	Balance
A	150
B	100

Question:
Now, when system crash
Occurs how the database
Can be restored back to original
Data values

Recovery using Shadowing

- Transferring Rs.100/- from account A to account B



ACCOUNTS

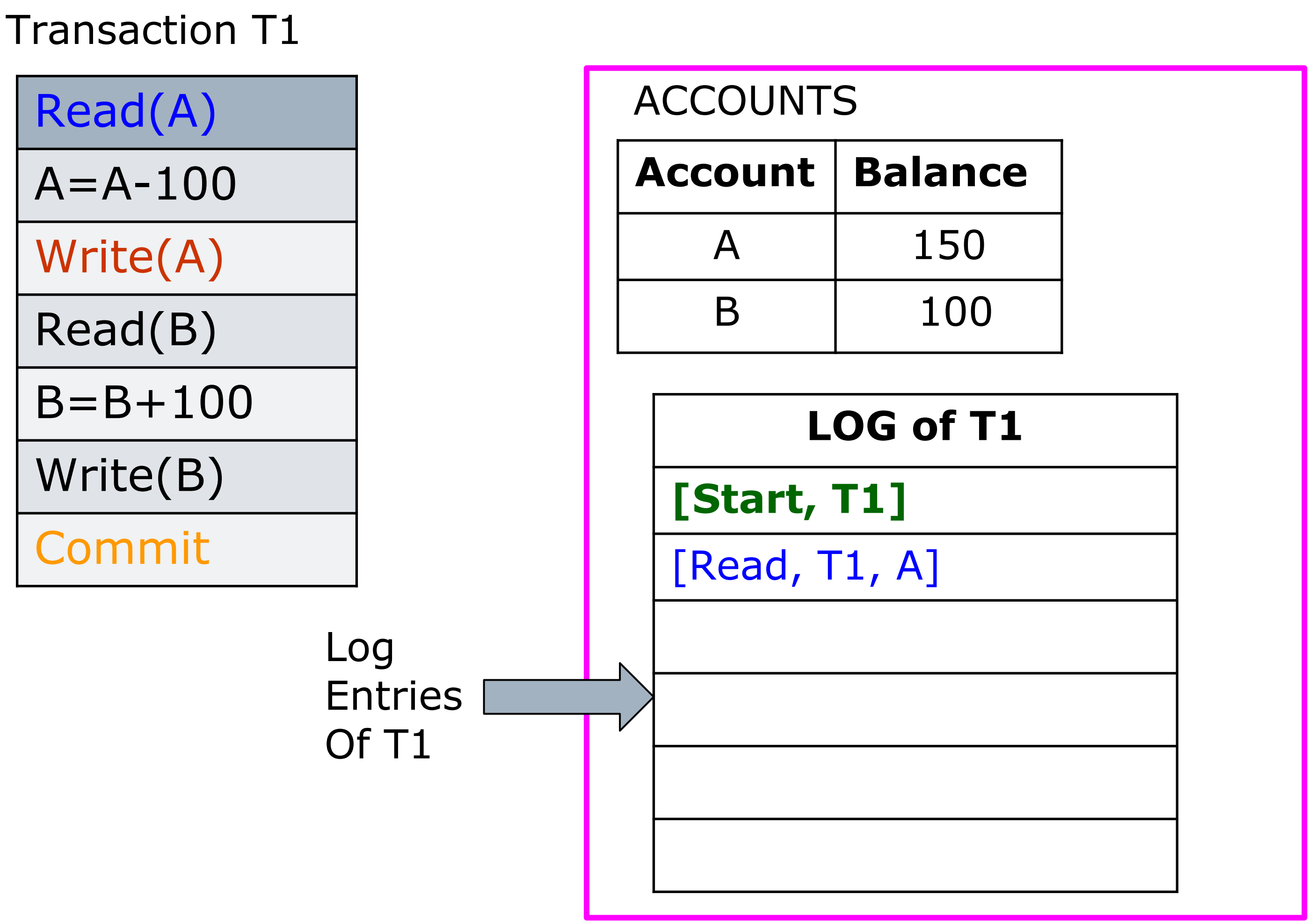
Account	Balance
A	~~150~~ ~~50~~ 150
B	100

Copy of ACCOUNTS

Account	Balance
A	150
B	100

Recovery using Log Entries

- Transferring Rs.100/- from account A to account B



Transaction T1

Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit

ACCOUNTS

Account	Balance
A	150
B	100

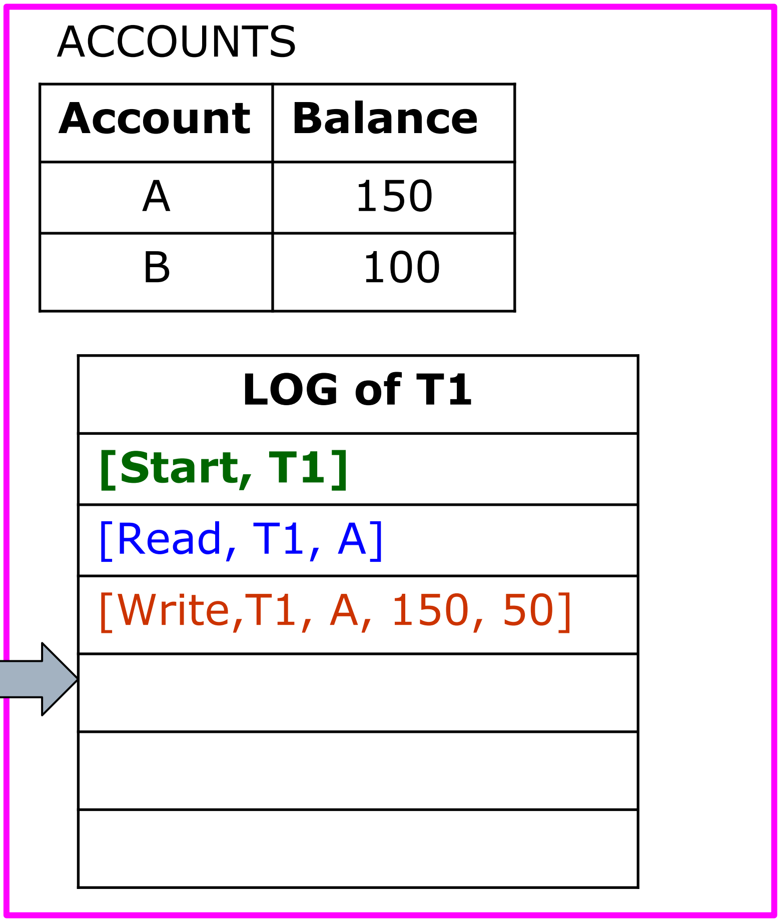
LOG of T1
[Start, T1]
[Read, T1, A]

Recovery using Log Entries

- Transferring Rs.100/- from account A to account B

Transaction T1

Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit



Account	Balance
A	150
B	100

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, 150, 50]

Recovery using Log Entries

- Transferring Rs.100/- from account A to account B

Transaction T1

Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit

Log Entries Of T1

ACCOUNTS

Account	Balance
A	150
B	100

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, 150, 50]
[Read, T1, B]
[Write, T2, B, 100,200]
[Commit, T1]

Recovery using Log Entries

- Transferring Rs.100/- from account A to account B

Transaction T1

Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit

System Crash →

Question:
Now, when system crash
Occurs how the database
Can be restored back to original
Data values

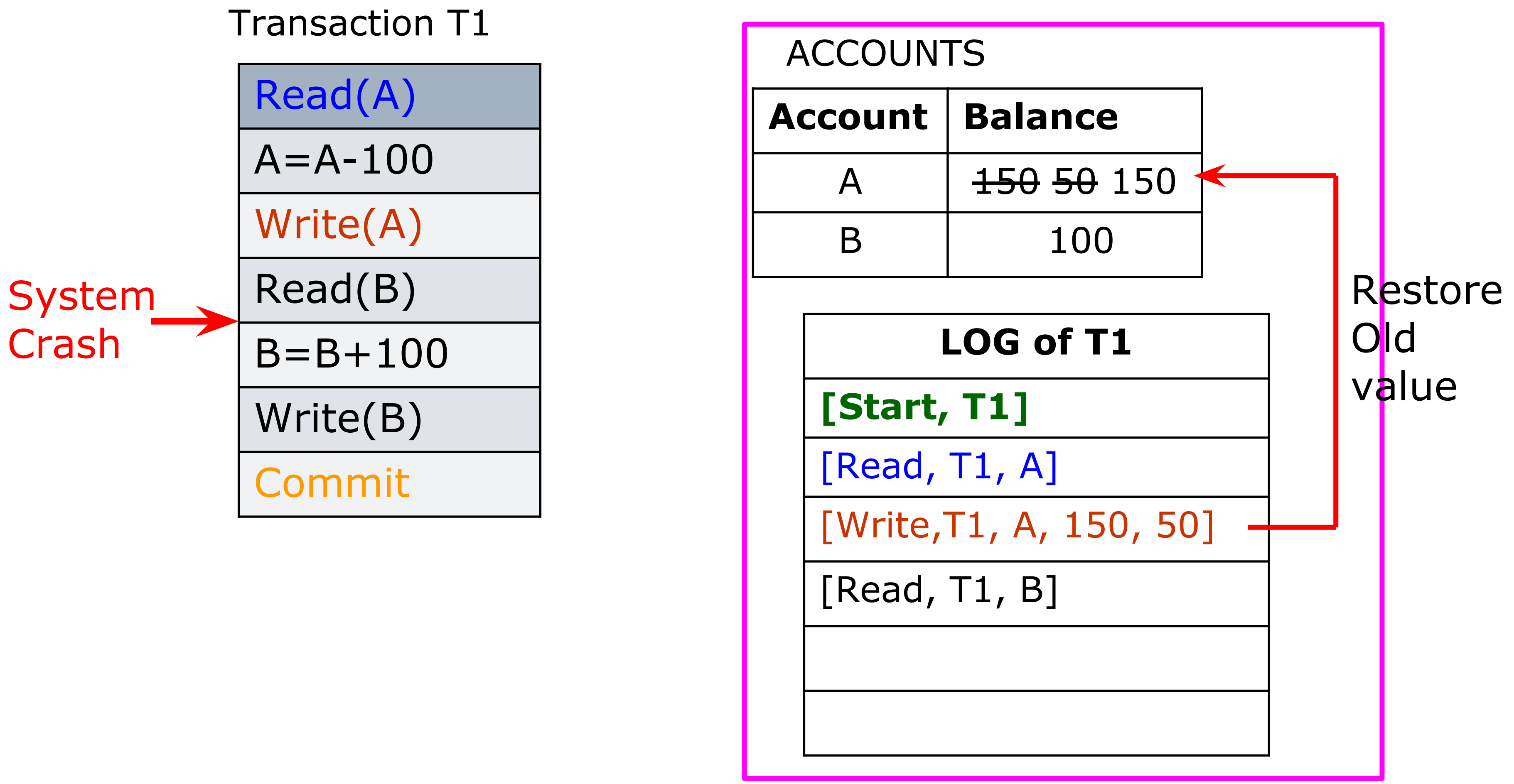
ACCOUNTS

Account	**Balance**
A	~~150~~ 50
B	100

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, 150, 50]
[Read, T1, B]

Recovery using Log Entries

- Transferring Rs.100/- from account A to account B



Transaction T1

Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit

ACCOUNTS

Account	Balance
A	~~150~~ ~~50~~ 150
B	100

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, 150, 50]
[Read, T1, B]

Recovery using Log Entries

- Transferring Rs.100/- from account A to account B

Transaction T1

Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit

ACCOUNTS

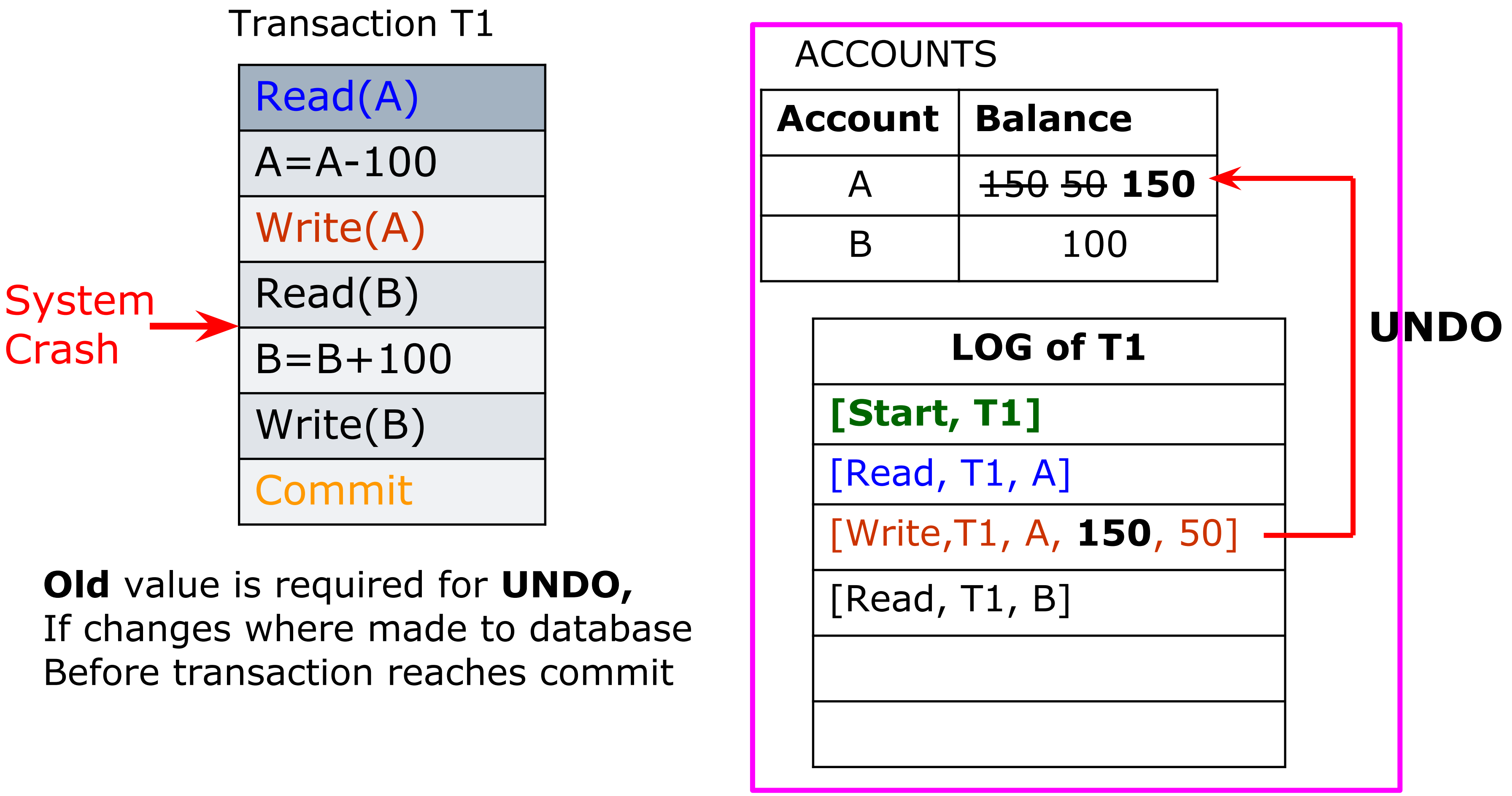
Account	**Balance**
A	150
B	100

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, 150, 50]

Question:
Why should we maintain old value in LOG for write operation

Recovery using Log Entries

- Transferring Rs.100/- from account A to account B



Transaction T1
Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit

ACCOUNTS

Account	Balance
A	~~150~~ ~~50~~ **150**
B	100

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, **150**, 50]
[Read, T1, B]

Old value is required for **UNDO,**
If changes where made to database
Before transaction reaches commit

Recovery using Log Entries

Consider two transactions, Transferring Rs.100/- from account A to account B and Rs. 200/- from account C to D

T1
Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit
T2
Read(C)
C=C-200
Write(C)
Read(D)
D=D+200
Write(D)
Commit

Account	Balance
A	150
B	100
C	400
D	200

Recovery using Log Entries

Consider two transactions, Transferring Rs.100/- from account A to account B and Rs. 200/- from account C to D

T1
Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit
T2
Read(C)
C=C-200
Write(C)
Read(D)
D=D+200
Write(D)
Commit

Account	Balance
A	~~150~~ 50
B	~~100~~ 200
C	~~400~~ 200
D	~~200~~ 400

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, 150, 50]
[Read, T1, B]
[Write, T1, B, 100,200]
[Commit, T1]
[Start, T2]
[Read, T2, C]
[Write,T2, C, 400, 200]
[Read, T2, D]
[Write, T2, D, 200,400]
[Commit, T2]

Recovery using Log Entries

Consider two transactions, Transferring Rs.100/- from account A to account B and Rs. 200/- from account C to D

T1
Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit
T2
Read(C)
C=C-200
Write(C)
Read(D)
D=D+200
Write(D)
Commit

System Crash $\rightarrow$ (between Read(D) and D=D+200)

Question:
Why we should maintain **New** value
in LOG for write operation

Account	Balance
A	150
B	100
C	400
D	200

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, 150, 50]
[Read, T1, B]
[Write, T1, B, 100,200]
[Commit, T1]
[Start, T2]
[Read, T2, C]
[Write,T2, C, 400, 200]
[Read, T2, D]

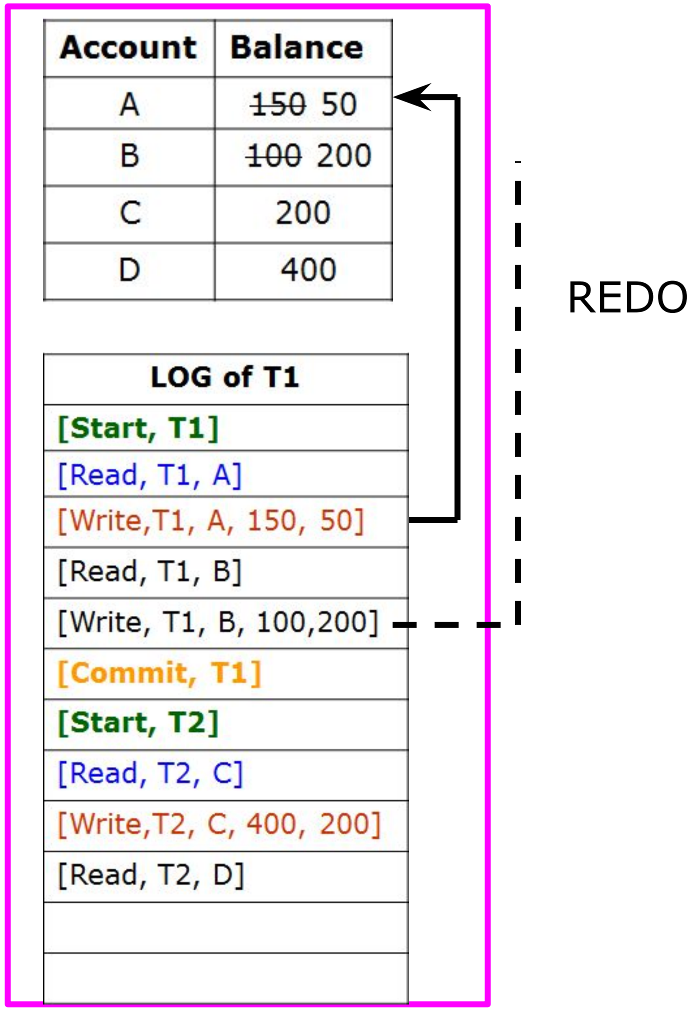
Recovery using Log Entries

Consider two transactions, Transferring Rs.100/- from account A to account B and Rs. 200/- from account C to D

T1
Read(A)
A=A-100
Write(A)
Read(B)
B=B+100
Write(B)
Commit
T2
Read(C)
C=C-200
Write(C)
Read(D)
D=D+200
Write(D)
Commit

System Crash

New value is required for **REDO** operation in
Case where you have
Reached commit point of
Transaction but
Changes where not
Updated
To database table



Account	Balance
A	~~150~~ 50
B	~~100~~ 200
C	200
D	400

LOG of T1
[Start, T1]
[Read, T1, A]
[Write,T1, A, 150, 50]
[Read, T1, B]
[Write, T1, B, 100,200]
[Commit, T1]
[Start, T2]
[Read, T2, C]
[Write,T2, C, 400, 200]
[Read, T2, D]

Recovery using Log Entries

1. Old value in LOG is required For UNDO operation

Old value is required for UNDO, If changes where made to database

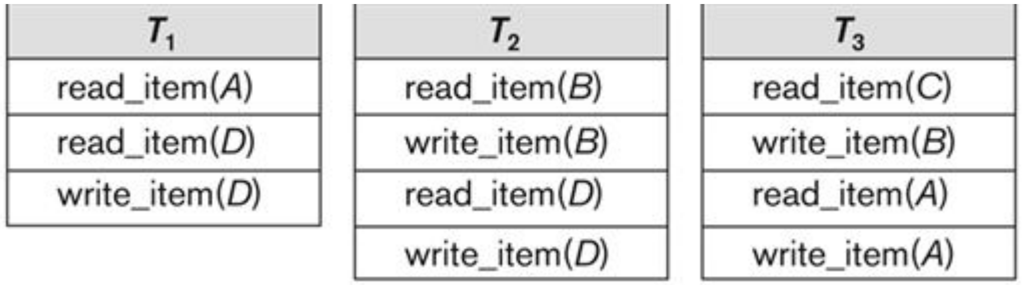
Before transaction reaches commit

2. New value in LOG is required For REDO operation

New value is required for REDO operation in Case where we have Reached commit point of Transaction but Changes where not Updated To database table.

Recovery using LOG enteries

Consider three transactions T1, T2, T3 as follows



T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

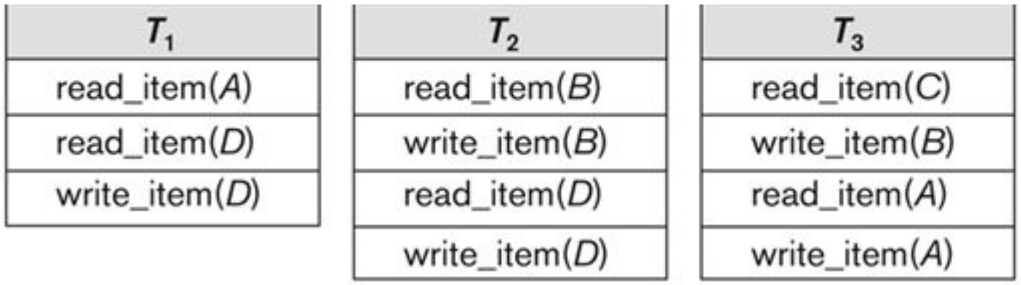
T_3
read_item(C)
write_item(B)
read_item(A)
write_item(A)

Database

A	30
B	15
C	40
D	20

Recovery using LOG enteries

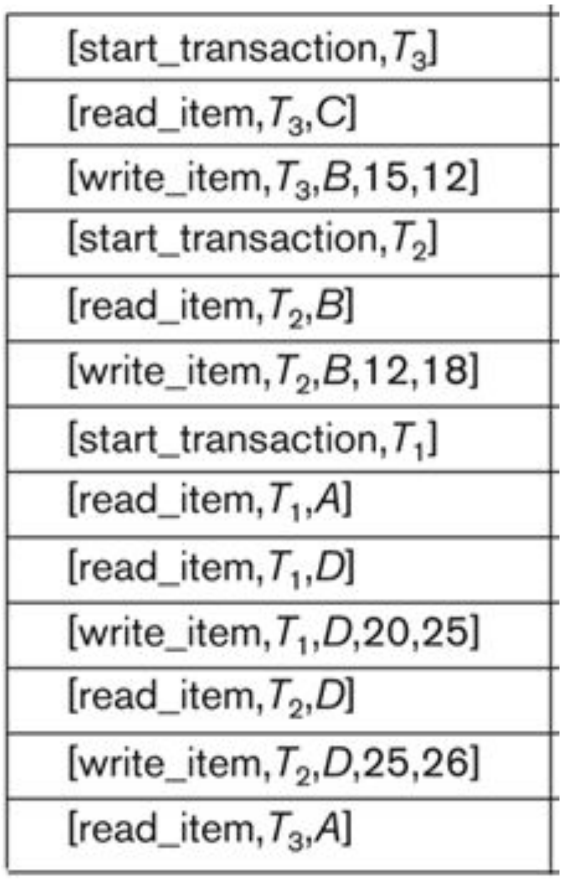
Consider three transactions T1, T2, T3 as follows



T_1	T_2	T_3
read_item(A)	read_item(B)	read_item(C)
read_item(D)	write_item(B)	write_item(B)
write_item(D)	read_item(D)	read_item(A)
	write_item(D)	write_item(A)

Database

A	30
B	15
C	40
D	20



[start_transaction,T_3]
[read_item,T_3,C]
[write_item,T_3,B,15,12]
[start_transaction,T_2]
[read_item,T_2,B]
[write_item,T_2,B,12,18]
[start_transaction,T_1]
[read_item,T_1,A]
[read_item,T_1,D]
[write_item,T_1,D,20,25]
[read_item,T_2,D]
[write_item,T_2,D,25,26]
[read_item,T_3,A]

Recovery using LOG enteries

Consider three transactions T1, T2, T3 as follows

Database

A	30
B	~~15~~ ~~12~~ 18
C	40
D	~~20~~ ~~25~~ 26

	A	*B*	*C*	*D*
	30	15	40	20
[start_transaction,T_3]				
[read_item,T_3,C]				
[write_item,T_3,B,15,12]		12		
[start_transaction,T_2]				
[read_item,T_2,B]				
[write_item,T_2,B,12,18]		18		
[start_transaction,T_1]				
[read_item,T_1,A]				
[read_item,T_1,D]				
[write_item,T_1,D,20,25]				25
[read_item,T_2,D]				
[write_item,T_2,D,25,26]				26
[read_item,T_3,A]				

System Crash $\rightarrow$

Recovery using LOG enteries

Consider three transactions T1, T2, T3 as follows

Database

A	30
B	~~15~~ ~~12~~ 18
C	40
D	~~20~~ ~~25~~ 26

	A	B	C	D
	30	15	40	20
[start_transaction, T_3]				
[read_item, T_3, C]				
[write_item, T_3, B, 15, 12]		12		
[start_transaction, T_2]				
[read_item, T_2, B]				
[write_item, T_2, B, 12, 18]		18		
[start_transaction, T_1]				
[read_item, T_1, A]				
[read_item, T_1, D]				
[write_item, T_1, D, 20, 25]				25
[read_item, T_2, D]				
[write_item, T_2, D, 25, 26]				26
[read_item, T_3, A]				

System Crash $\longrightarrow$

Question:
After System, how can we recovery Database i.e
B value to 15,
D value to 20

Recovery using LOG enteries

Consider three transactions T1, T2, T3 as follows

* T3 transaction is rolled back because it did not reached its commit point
** T2 is rolled back because it reads the value of item B written by T3

Database

A	30
B	~~15~~ ~~12~~ ~~18~~ 15
C	40
D	~~20~~ ~~25~~ 26

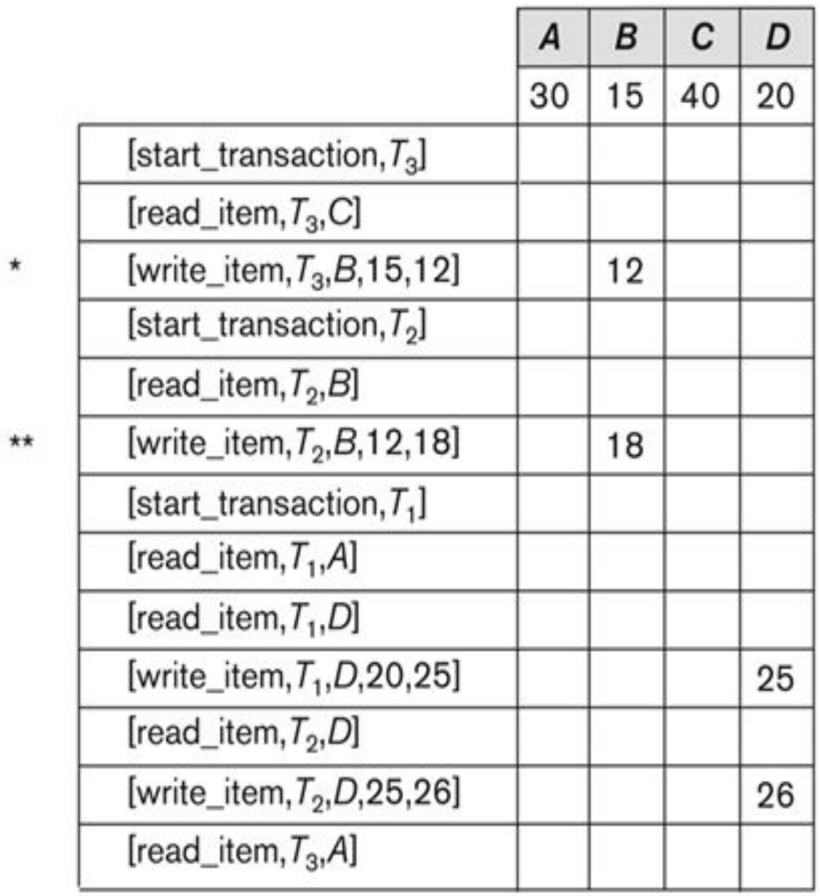
		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
	[read_item,T_2,D]				
	[write_item,T_2,D,25,26]				26
	[read_item,T_3,A]				

System Crash $\rightarrow$

Recovery using LOG enteries

Consider three transactions T1, T2, T3 as follows

Question:
Can you answer now, why we maintain
Data item name and transaction number
For read operation in LOG ?



		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
	[read_item,T_2,D]				
	[write_item,T_2,D,25,26]				26
	[read_item,T_3,A]				

System Crash $\rightarrow$

Database

A	30
B	~~15~~ ~~12~~ 18
C	40
D	~~20~~ ~~25~~ 26

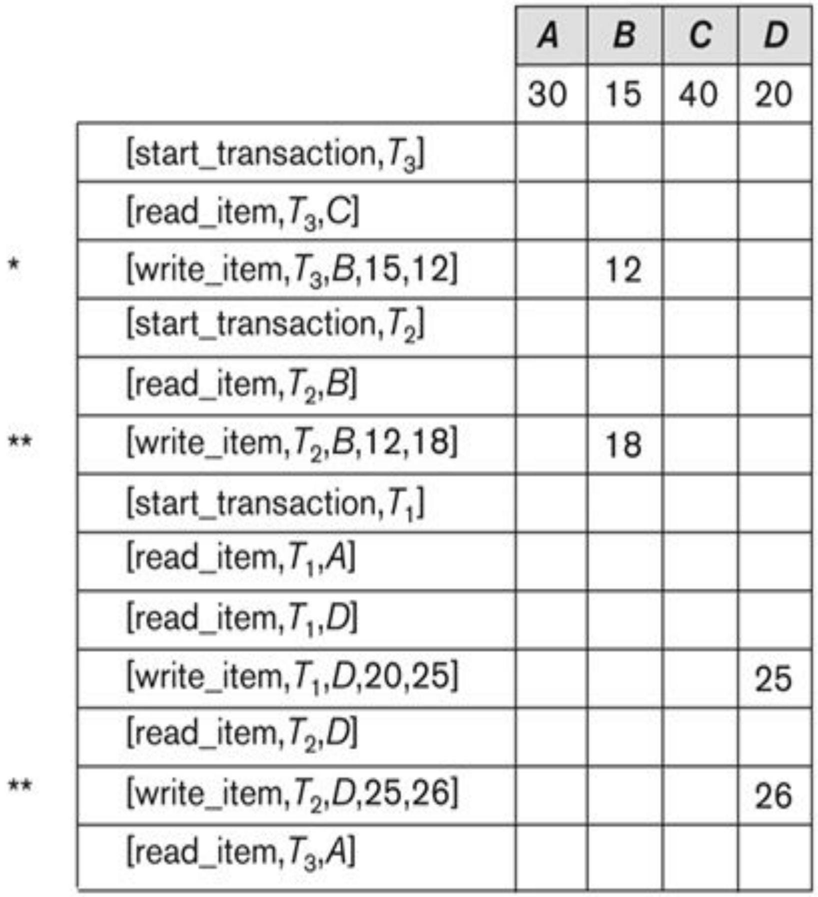
Recovery using LOG enteries

Consider three transactions T1, T2, T3 as follows

T3 and T1 transaction is rolled back because it did not reached its commit point
T2 is rolled back because it reads the value of item
- B written by T3
- D written by T1

Database

A	30
B	~~15~~ ~~12~~ ~~18~~ 15
C	40
D	~~20~~ ~~25~~ ~~26~~ 20



		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
	[read_item,T_2,D]				
**	[write_item,T_2,D,25,26]				26
	[read_item,T_3,A]				

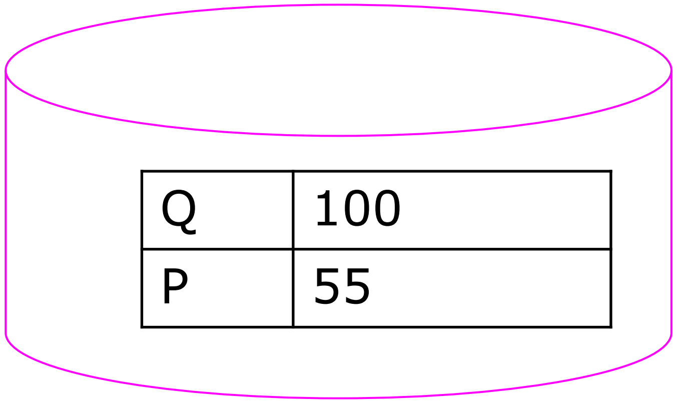
System Crash →

Problem to Solve

- Consider the following example of log for two transactions.

1. (Start, T1);
2. (Write, T1, Q, 100,50);
3. (Commit, T1);
4. (Start, T2);
5. (Write, T2, P, 55, 10);
6. (Commit, T2);

Consider the case where the schedule crashes after Step 4 and before Step 5, then the question is which operation should we REDO if following is the scenario of database just before crash.



Q	100
P	55

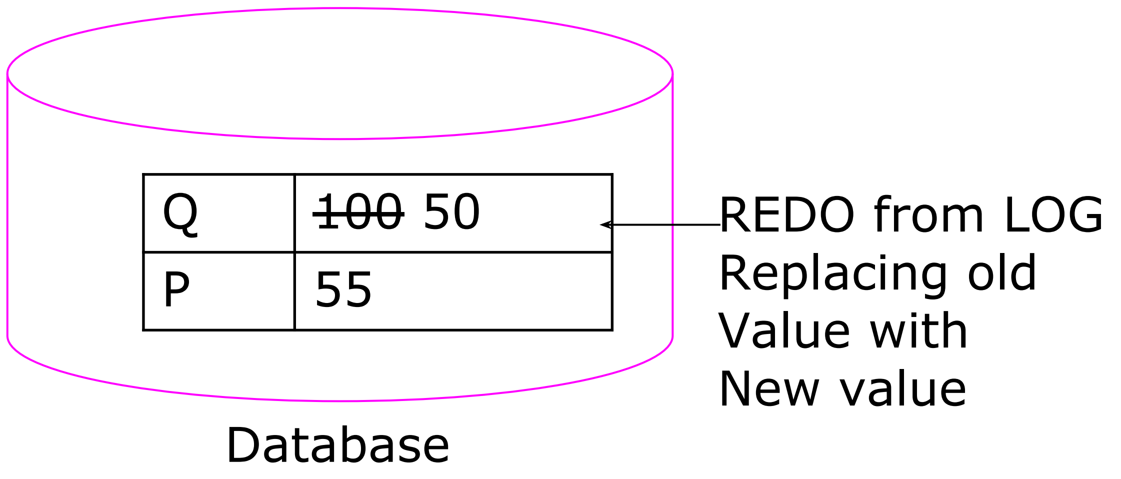
Database

Answer

- Consider the following example of log for two transactions.

1. (Start, T1);
2. (Write, T1, Q, 100,50);
3. (Commit, T1);
4. (Start, T2);
5. (Write, T2, P, 55, 10);
6. (Commit, T2);

Consider the case where the schedule crashes after Step 4 and before Step 5, then the question is which operation should we REDO if following is the scenario of database just before crash.

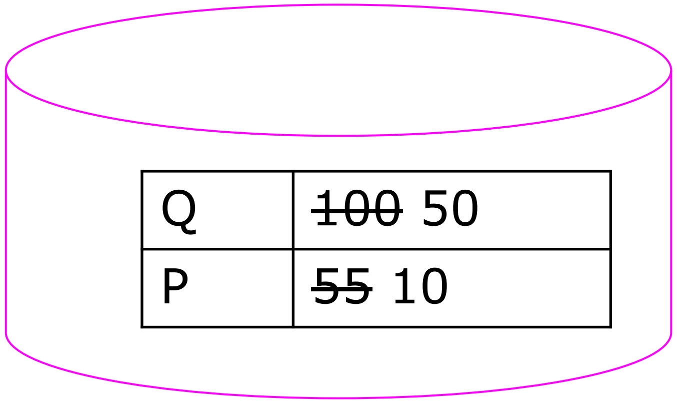


Problem to Solve

- Consider the following example of log for two transactions.

1. (Start, T1);
2. (Write, T1, Q, 100,50);
3. (Commit, T1);
4. (Start, T2);
5. (Write, T2, P, 55, 10);
6. (Commit, T2);

Consider the case where the schedule crashes after Step 5 and before Step 6, then the question is which operation should we UNDO if following is the scenario of database just before crash.



Q	~~100~~ 50
P	~~55~~ 10

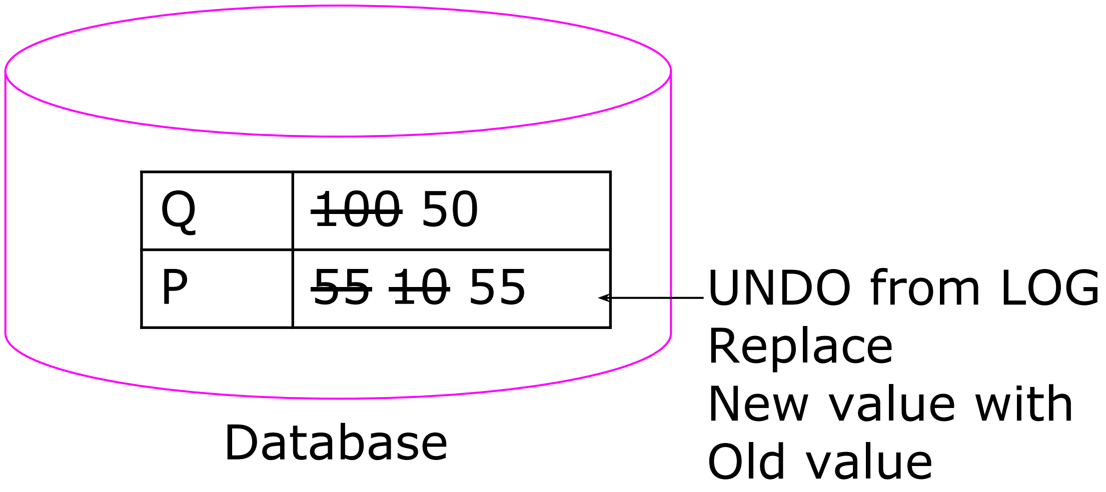
Database

Answer

- Consider the following example of log for two transactions.

1. (Start, T1);
2. (Write, T1, Q, 100,50);
3. (Commit, T1);
4. (Start, T2);
5. (Write, T2, P, 55, 10);
6. (Commit, T2);

Consider the case where the schedule crashes after Step 5 and before Step 6, then the question is which operation should we UNDO if following is the scenario of database just before crash.



Approaches to Recovery

Two Types using LOG enteries

1. Deferred Update
2. Immediate Update

Log-Based Recovery

- A log is kept on stable storage.
  - The log is a sequence of log records, and maintains a record of update activities on the database.
- When transaction Ti starts, it registers itself by writing a <Ti start> log record
- Before Ti executes write(X), a log record <write,Ti,X,V1,V2> is written, where V1 is the value of X before the write, and V2 is the new value to be written to X.
  - Log record notes that Ti has performed a write on data item X that had value V1 before the write, and will have value V2 after the write.
- When Ti finishes it last statement, the log record <Commit Ti> is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)

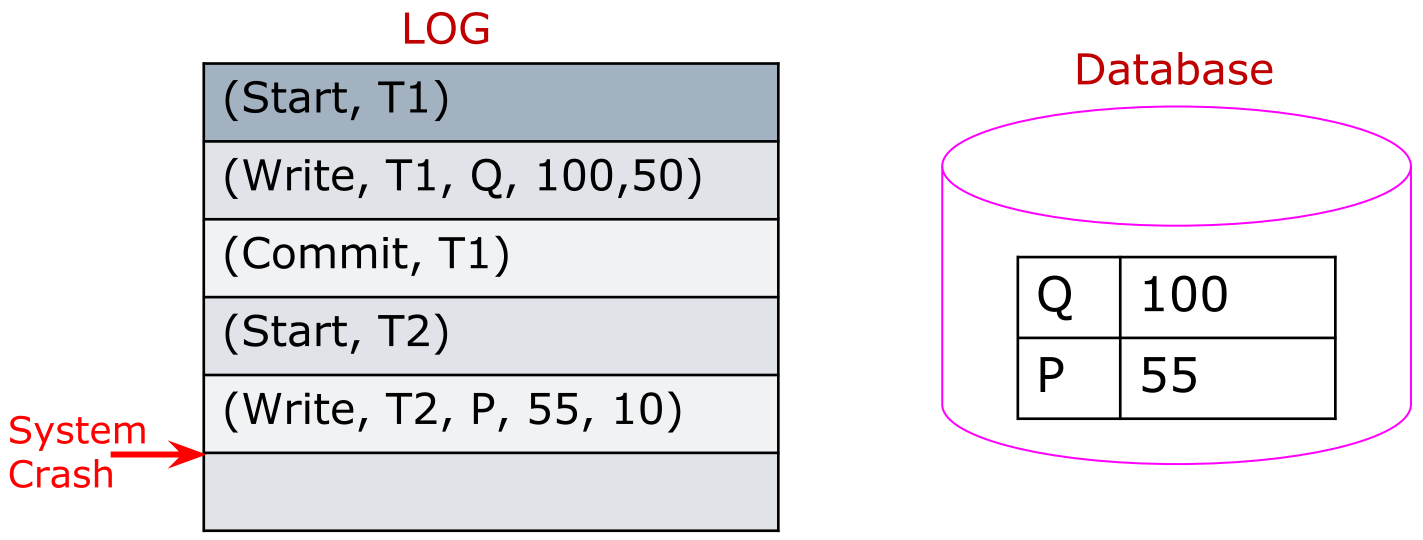
Deferred Database Modification

- The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing <start Ti> record to log.
- A write(X) operation results in a log record <write,Ti,X,V1,V2> being written, where V1 is old value of X, V2 is the new value for X
  - Note: old value V1 is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When Ti partially commits, <Commit, Ti> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.
- During recovery after a crash, a transaction needs to be redone if and only if both <start Ti> and <commit Ti> are there in the log.
- Redoing a transaction Ti (redoTi) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while: the transaction is executing the original updates or while recovery action is being taken
- Deferred modification is also referred as **No-Undo/Redo** method

Deferred Database Modification

Example of Deferred based recovery

- Consider log entries for two transactions T1 and T2 as follows.



Under deferred recovery method, after system crash the algorithm looks at LOG enteries for all transactions Ti if the entry of both <start, Ti> and <commit, Ti> is present then for all write operations REDO will be carried out i.e., replacing with new value

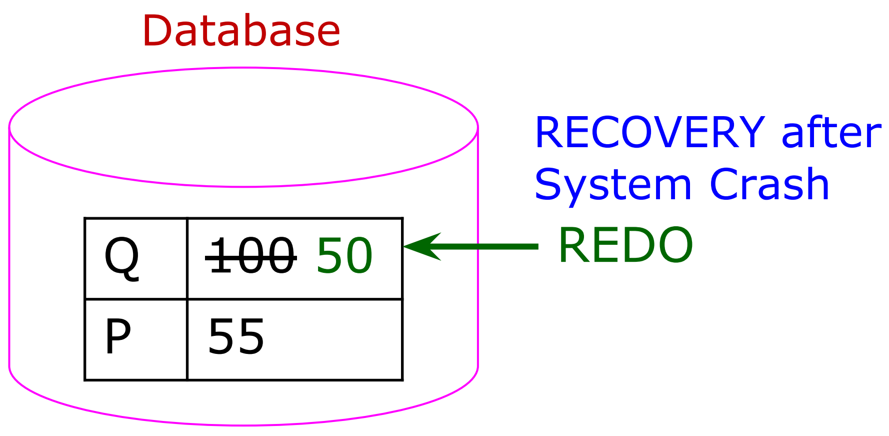
Deferred Database Modification

Example of Deferred based recovery

- Consider log entries for two transactions T1 and T2 as follows.

LOG
(Start, T1)
(Write, T1, Q, 100,50)
(Commit, T1)
(Start, T2)
(Write, T2, P, 55, 10)

System Crash $\rightarrow$

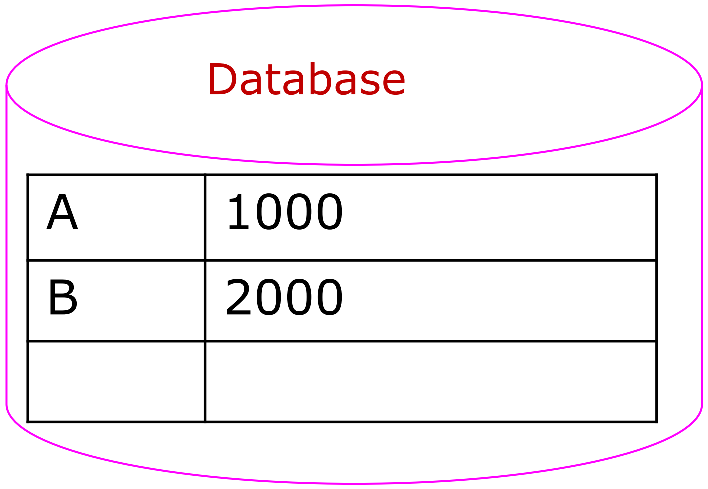


Under deferred recovery method, after system crash the algorithm looks at LOG enteries for all transactions Ti if the entry of both <start, Ti> and <commit, Ti> is present then for all write operations REDO will be carried out i.e., replacing with new value

Deferred Database Modification

Question:
If following is the LOG on stable storage after systems crash, then recovery algorithm should carry out any REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>



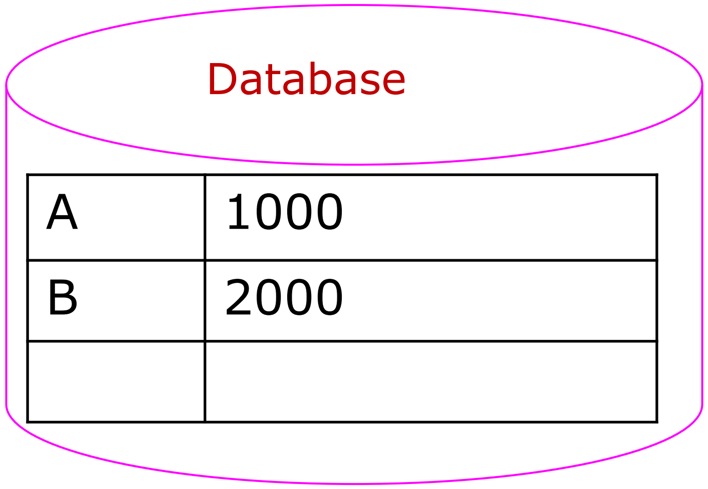
A	1000
B	2000

Deferred Database Modification

Question:

If following is the LOG on stable storage after systems crash, then recovery algorithm should carry out any REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>



A	1000
B	2000

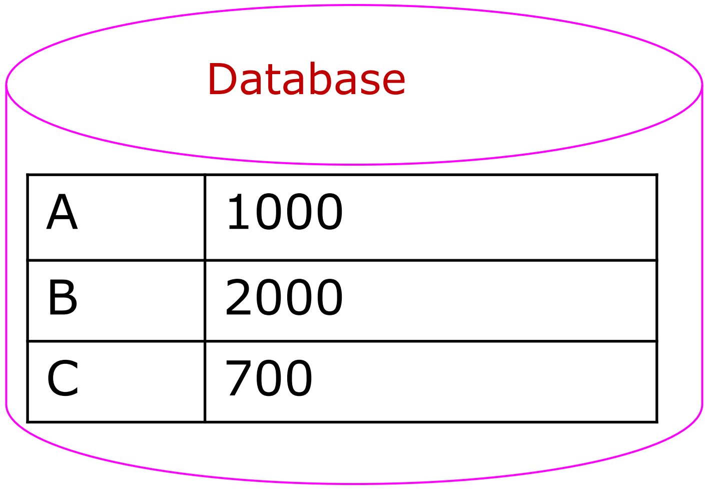
Answer:

No REDO operation need to be taken because no commit LOG entry found for any transaction

Deferred Database Modification

Question:
If following is the LOG on stable storage after systems crash, then recovery algorithm should carry out any REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>
<Commit, T1>
<Start, T2>
<write,T2,C,700,600>

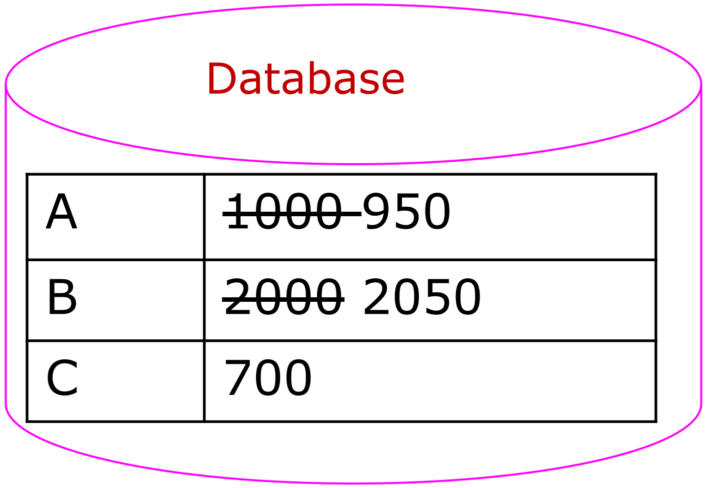


A	1000
B	2000
C	700

Deferred Database Modification

Question:
If following is the LOG on stable storage after systems crash, then recovery algorithm should carry out any REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>
<Commit, T1>
<Start, T2>
<write,T2,C,700,600>


Database

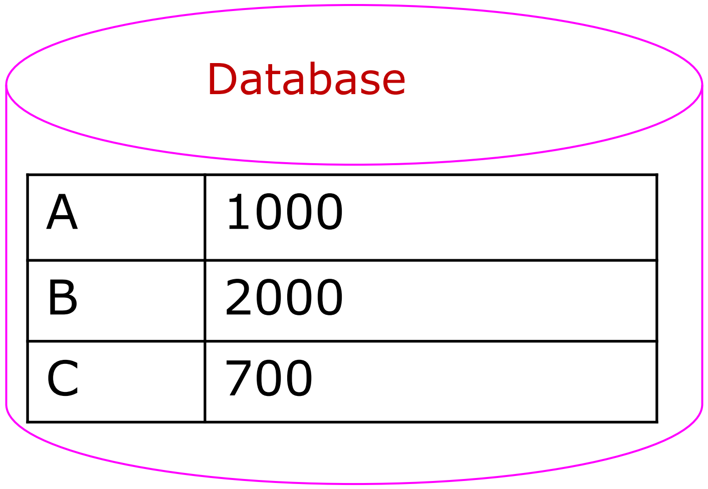
A	~~1000~~ 950
B	~~2000~~ 2050
C	700

Answer:
REDO operation for transaction T1 to data items A and B should be performed because commit LOG entry found for the transaction T1 <Commit, T1>

Deferred Database Modification

Question:
If following is the LOG on stable storage after systems crash, then recovery algorithm should carry out any REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>
<Commit, T1>
<Start, T2>
<write,T2,C,700,600>
<Commit, T2>

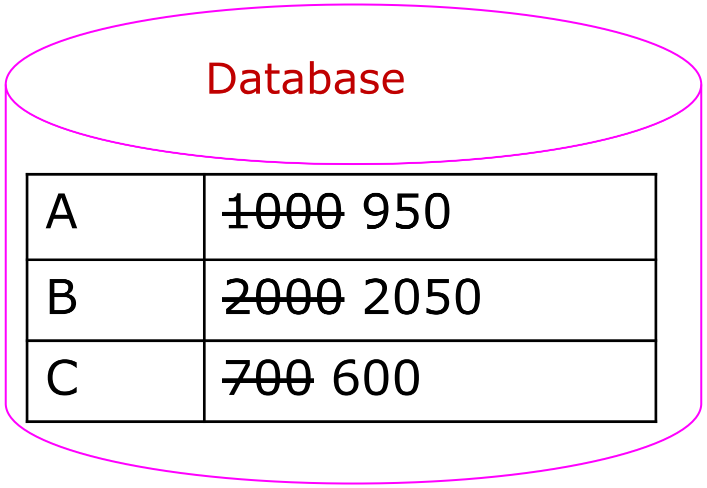


A	1000
B	2000
C	700

Deferred Database Modification

Question:
If following is the LOG on stable storage after systems crash, then recovery algorithm should carry out any REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>
<Commit, T1>
<Start, T2>
<write,T2,C,700,600>
<Commit, T2>



A	~~1000~~ 950
B	~~2000~~ 2050
C	~~700~~ 600

Answer:
REDO operation for Transaction T1 must be performed followed by the transaction T2 since <Commit, T1> and <Commit,T2> are present.

Immediate Database Modification

- The immediate database modification scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written before database item is written
- We assume that the log record is output directly to stable storage
- Can be extended to postpone log record output, so as long as prior to execution of an output(B) operation for a data block B, all log records corresponding to items B must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification

- Recovery procedure has two operations instead of one:
  - **undo(Ti)** restores the value of all data items updated by Ti to their **old values**, going backwards from the last log record for Ti
  - **redo(Ti)** sets the value of all data items updated by Ti to the **new values**, going forward from the first log record for Ti
- Both operations must be idempotent
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction Ti needs to be **undone** if the log contains the <start, Ti> record , but **does not contain the <commit, Ti>** record .
  - Transaction Ti needs to be **redone** if the log contains **both the <start, Ti> record and <commit, Ti>** the record .
- Undo operations are performed first, then redo operations.

Immediate Database Modification

Two main categories of Immediate update algorithm

UNDO/No-REDO

- If the recovery technique ensures that all updates of a transaction are recorded in the database on disk before the transaction commits, there is never a need to redo any operations of committed transactions. Such an algorithm is called undo/no-redo.

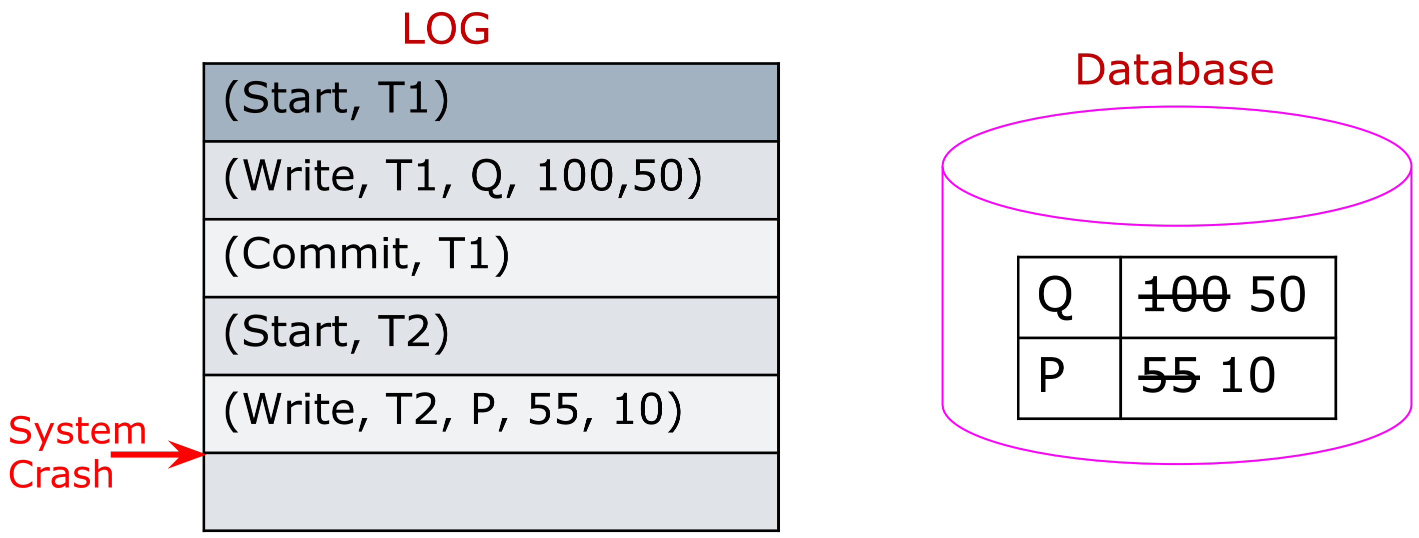
UNDO/REDO

- On the other hand, if the transaction is allowed to commit before all its changes are written to the database, we have the undo/redo method, the most general recovery algorithm.

Immediate Database Modification

Example of Immediate database recovery

- Consider log entries for two transactions T1 and T2 as follows.



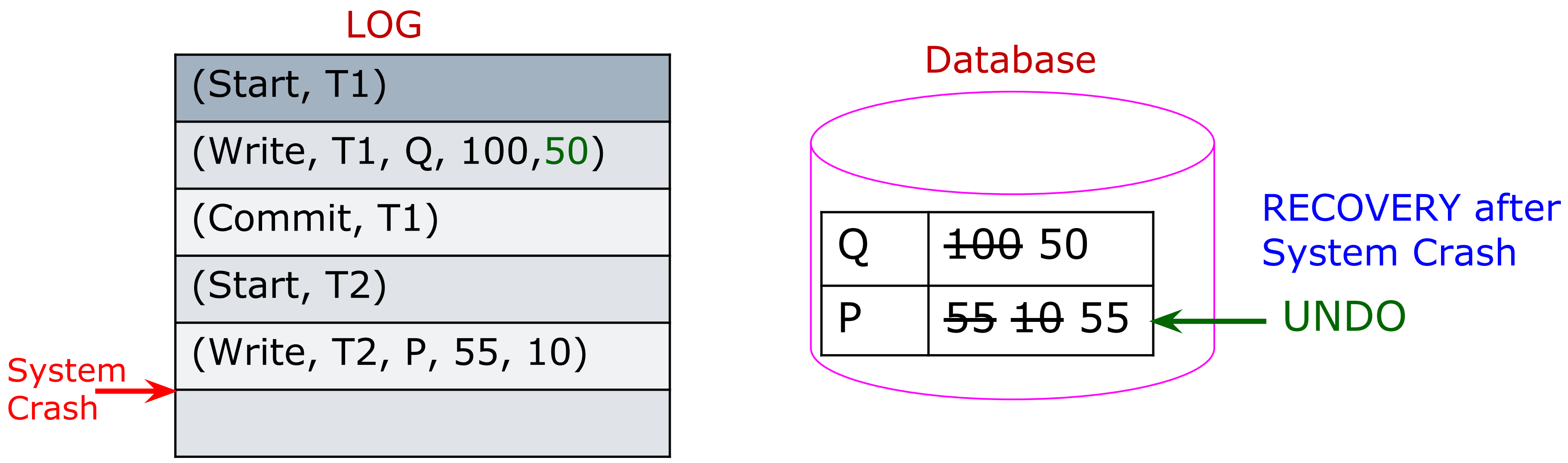
Under immediate recovery method, after system crash the algorithm looks at LOG enteries for all transactions Ti

- if the entry of both <start, Ti> and <commit, Ti> is present then for all write operations of Ti REDO will be carried out if updates have not been carried out on stable storage of database
- if the entry of only <start, Ti> is present but not <commit, Ti> then for all write operations of Ti UNDO will be carried out

Deferred Database Modification

Example of Deferred based recovery

- Consider log entries for two transactions T1 and T2 as follows.



LOG

(Start, T1)
(Write, T1, Q, 100,50)
(Commit, T1)
(Start, T2)
(Write, T2, P, 55, 10)

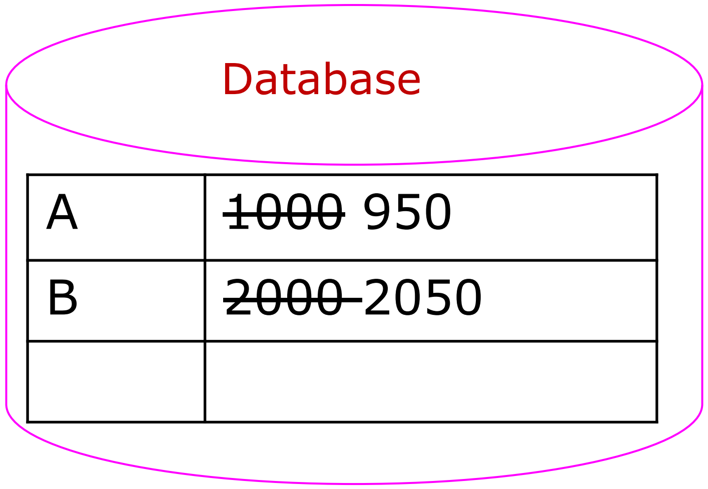
Under immediate recovery method, after system crash the algorithm looks at LOG enteries for all transactions Ti
- if the entry of both <start, Ti> and <commit, Ti> is present then for all write operations of Ti REDO will be carried out if updates have not been carried out on stable storage of database
- if the entry of only <start, Ti> is present but not <commit, Ti> then for all write operations of Ti UNDO will be carried out

Immediate Database Modification

Question:

If following is the LOG and database on stable storage after systems crash, then recovery algorithm should carry out any UNDO/REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>



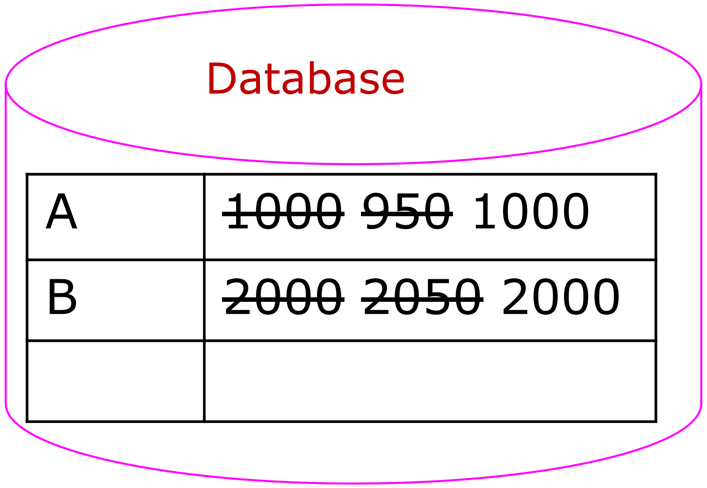
A	~~1000~~ 950
B	~~2000~~ 2050

Immediate Database Modification

Question:

If following is the LOG and database on stable storage after systems crash, then recovery algorithm should carry out any UNDO/REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>



A	~~1000~~ ~~950~~ 1000
B	~~2000~~ ~~2050~~ 2000

Answer:

UNDO for T1 because <commit,T1> not found: A should be restored to 1000 and B should be restored to 2000

Immediate Database Modification

Question:
If following is the LOG and database on stable storage after systems crash, then recovery algorithm should carry out any UNDO/REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>
<Commit, T1>
<Start, T2>
<write,T2,C,700, 600>

Database

A	~~1000~~ 950
B	~~2000~~ 2050
C	~~700~~ 600

Immediate Database Modification

Question:

If following is the LOG and database on stable storage after systems crash, then recovery algorithm should carry out any UNDO/REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>
<Commit, T1>
<Start, T2>
<write,T2,C,700, 600>

Database

A	~~1000~~ 950
B	~~2000~~ 2050
C	~~700~~ ~~600~~ 700

Answer:

Recovery actions

UNDO T2 because <commit, T2> not found: C is restored to 700

Immediate Database Modification

Question:

If following is the LOG and database on stable storage after systems crash, then recovery algorithm should carry out any UNDO/REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>
<Commit, T1>
<Start, T2>
<write,T2,C,700, 600>
<Commit, T2>

Database

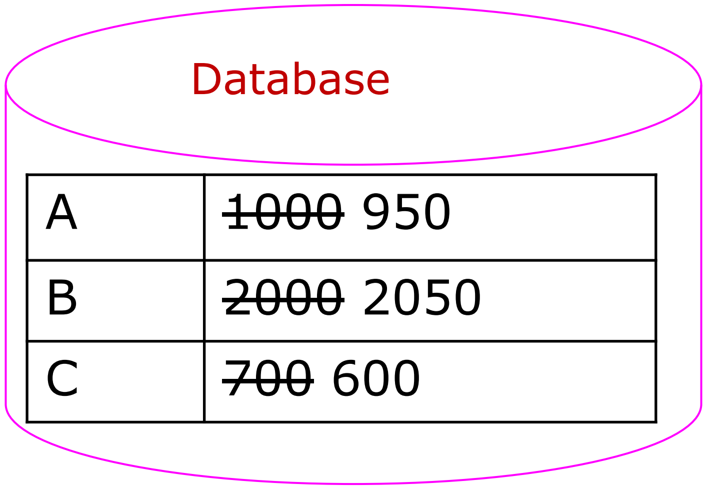
A	1000
B	2000
C	700

Immediate Database Modification

Question:

If following is the LOG and database on stable storage after systems crash, then recovery algorithm should carry out any UNDO/REDO operation ?

<Start, T1>
<write,T1,A,1000,950>
<write,T1,B,2000,2050>
<Commit, T1>
<Start, T2>
<write,T2,C,700, 600>
<Commit, T2>



A	~~1000~~ 950
B	~~2000~~ 2050
C	~~700~~ 600

Answer:

Recovery actions

REDO operation for Transaction T1 and T2 since <Commit, T1> and <Commit,T2> are present.

Checkpoints

Problems in recovery procedure as discussed earlier :

1. Searching the entire log is time-consuming
2. We might unnecessarily redo transactions which have already output their updates to the database.

Streamline recovery procedure by periodically performing **checkpointing**

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk.
3. Write a log record <**checkpoint**> onto stable storage.

Checkpoints

During recovery we need to consider only the most recent transaction Ti that started before the checkpoint, and transactions that started after *Ti*.

1. Scan backwards from end of log to find the most recent <**checkpoint**> record

2. Continue scanning backwards till a record <**start** *Ti*> is found.

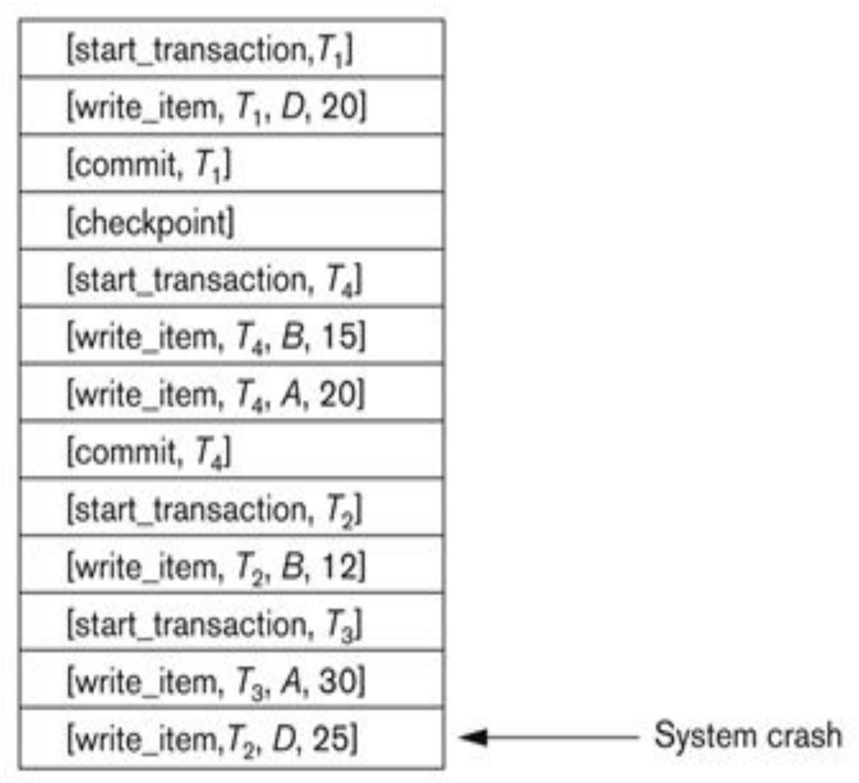
3.Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.

4.For all transactions (starting from *Ti* or later) with no < **commit** *Ti*>, execute **undo***(Ti).* (Done only in case of immediate modification.)

5.Scanning forward in the log, for all transactions starting from *Ti* or later with a <**commit** *Ti*>, execute **redo***(Ti).*

Recovery using Deferred Update with Concurrent Transactions

Consider the following LOG enteries



[start_transaction, T_1]
[write_item, T_1, D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, B, 15]
[write_item, T_4, A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2, B, 12]
[start_transaction, T_3]
[write_item, T_3, A, 30]
[write_item, T_2, D, 25]

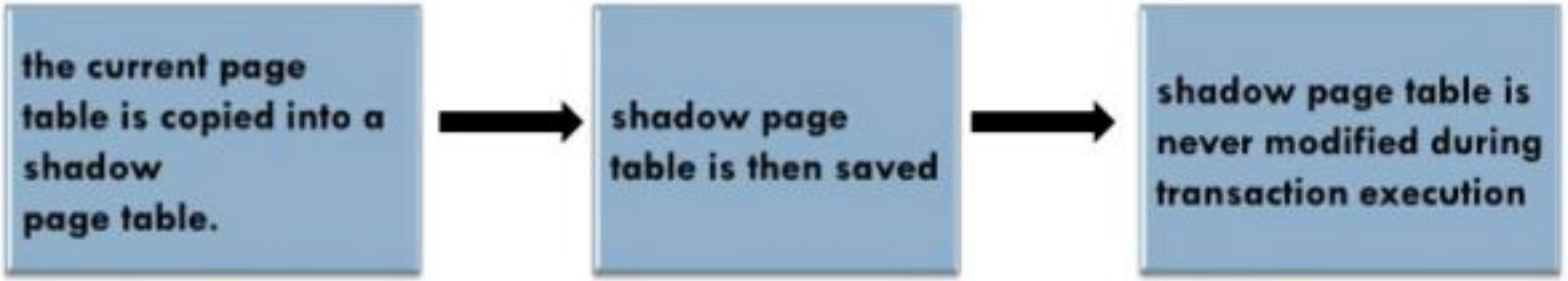
System crash

T2 and T3 are ignored because they did not reach their commit points
T4 is redone because its commit point is after the last system check point

Shadow Paging: Recovery Scheme that does not require log and useful for single-user environment

- In this technique, the database is considered to be made up of fixed-size disk blocks or pages for recovery purposes.
- Maintains two tables during the lifetime of a transaction-current page table and shadow page table.
- Store the shadow page table in nonvolatile storage, to recover the state of the database prior to transaction execution
- This is a technique for providing atomicity and durability.

When a transaction begins executing



the current page table is copied into a shadow page table. → shadow page table is then saved → shadow page table is never modified during transaction execution

Shadow Paging

Hard Disk

Database table

Page No.	Page/ Sub-table Data	Location
1	Data Item **A** 1000	Addr1
2	**B** 2000	Addr2
3	**C** 700	Addr3
4	**D** 50	Addr4

Shadow Paging

Hard Disk

Database table

Page No.	Page/ Sub-table Data	Location
1	Data Item **A** 1000	Addr1
2	**B** 2000	Addr2
3	**C** 700	Addr3
4	**D** 50	Addr4

Transaction operations

```
(Start,T1)
(Write,T1,A,1000,950)
(Write,T1,B,2000,2050)
(Commit,T1)
(Start, T2)
(Write,T2,C,700,600)
(Write,T2,D, 50,30)
```

Shadow Paging

Hard Disk

Main Memory

Current Page Directory

Page 1: Addr1
Page 2: Addr2
Page 3: Addr3
Page 4: Addr4

Database table

Page No.	Page/ Sub-table Data	Location
1	Data Item A 1000	Addr1
2	B 2000	Addr2
3	C 700	Addr3
4	D 50	Addr4

Transaction operations

(Start,T1)
(Write,T1,A,1000,950)
(Write,T1,B,2000,2050)
(Commit,T1)
(Start, T2)
(Write,T2,C,700,600)
(Write,T2,D,50,30)

When transaction starts Current Page directory is kept in main memory and Shadow directory on disk

Shadow Page Directory

Page 1: Addr1
Page 2: Addr2
Page 3: Addr3
Page 4: Addr4

Shadow Paging

Hard Disk

Database table

Page No.	Page/ Sub-table Data	Location
1	Data Item A 1000	Addr1
2	B 2000	Addr2
3	C 700	Addr3
4	D 50	Addr4
5	A 950	Addr5

Shadow Page Directory

Page 1: Addr1
Page 2: Addr2
Page 3: Addr3
Page 4: Addr4

Main Memory

Current Page Directory

Page 1: ~~Addr1~~ Addr5
Page 2: Addr2
Page 3: Addr3
Page 4: Addr4

Transaction operations

(Start,T1)
(Write,T1,A,1000,950)
(Write,T1,B,2000,2050)
(Commit,T1)
(Start, T2)
(Write,T2,C,700,600)
(Write,T2,D,50,30)

After execution of (write,T1,A,1000,950) Then new page table is created on disk and Current page Directory updated. But shadow directory will not be updated

Shadow Paging

Hard Disk

Main Memory

Current Page Directory

Page 1: ~~Addr1~~ Addr5
Page 2: ~~Addr2~~ Addr6
Page 3: Addr3
Page 4: Addr4

Transaction operations

(Start,T1)
(Write,T1,A,1000,950)
(Write,T1,B,2000,2050)
(Commit,T1)
(Start, T2)
(Write,T2,C,700,600)
(Write,T2,D,50,30)

After execution of (write,T2,2000,2050) Then new page table is created on disk and Current page Directory updated. But shadow directory will not be updated

Database table

Page No.	Page/ Sub-table Data	Location
1	Data Item A 1000	Addr1
2	B 2000	Addr2
3	C 700	Addr3
4	D 50	Addr4
5	A 950	Addr5
6	B 2050	Addr6

Shadow Page Directory

Page 1: Addr1
Page 2: Addr2
Page 3: Addr3
Page 4: Addr4

Shadow Paging

Hard Disk

Main Memory

Current Page Directory

Page 1: ~~Addr1~~ Addr5
Page 2: ~~Addr2~~ Addr6
Page 3: Addr3
Page 4: Addr4

Transaction operations

(Start,T1)
(Write,T1,A,1000,950)
(Write,T1,B,2000,2050)
(Commit,T1)
(Start, T2)
(Write,T2,C,700,600)
(Write,T2,D,50,30)

After execution of (commit,T1)
Shadow directory will be updated and old page On the hard disk will be discarded

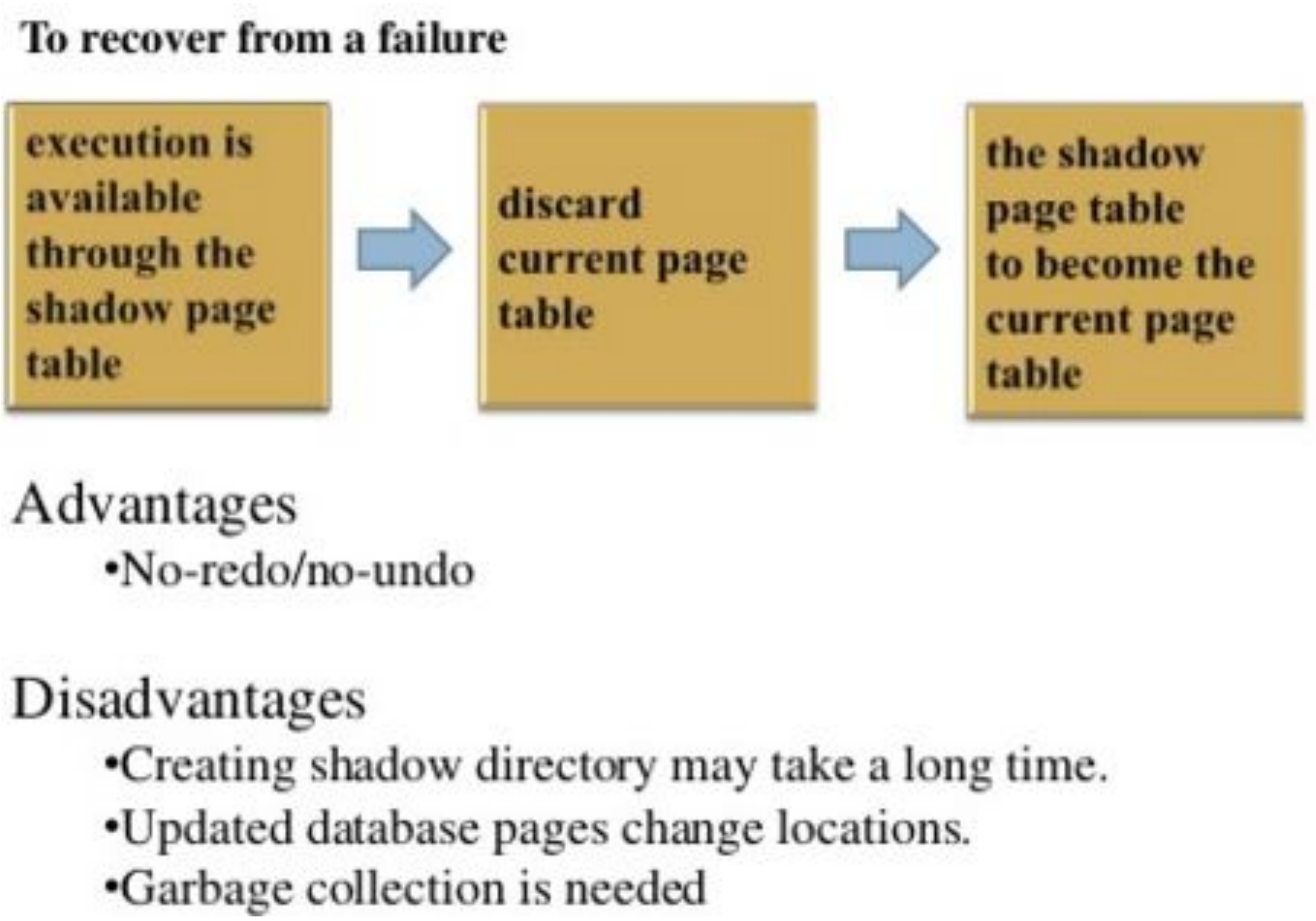
Database table

Page No.	Page/ Sub-table Data	Location
~~1~~	~~Data Item A 1000~~	~~Addr1~~
~~2~~	~~B 2000~~	~~Addr2~~
3	C 700	Addr3
4	D 50	Addr4
5	A 950	Addr5
6	B 2050	Addr6

Shadow Page Directory

Page 1: ~~Addr1~~ Addr5
Page 2: ~~Addr2~~ Addr6
Page 3: Addr3
Page 4: Addr4

Shadow Paging



To recover from a failure

execution is available through the shadow page table → **discard current page table** → **the shadow page table to become the current page table**

Advantages

- No-redo/no-undo

Disadvantages

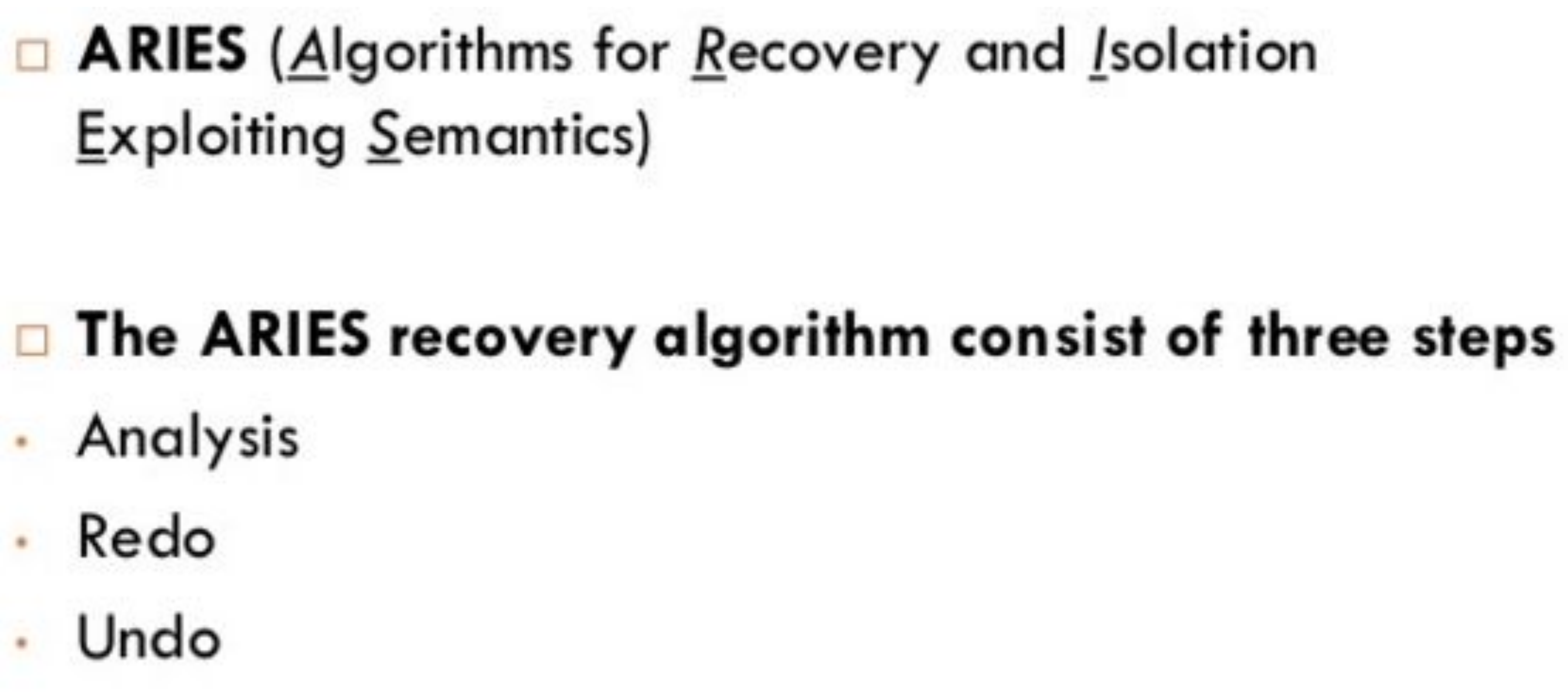
- Creating shadow directory may take a long time.
- Updated database pages change locations.
- Garbage collection is needed

“ARIES” recovery algorithm

Recovery algorithms are techniques to ensure database consistency ,transaction atomicity and durability without any failure.

- **Recovery algorithms have two parts**
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures.
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

“ARIES” recovery algorithm



- **ARIES** (Algorithms for Recovery and Isolation Exploiting Semantics)

- **The ARIES recovery algorithm consist of three steps**
  - Analysis
  - Redo
  - Undo

“ARIES” recovery algorithm

- **Analysis -** Identify the dirty pages(updated pages) in the buffer and set of active transactions at the time of failure.
- **Redo -** Re-apply updates from the log to the database. It will be done for the committed transactions.
- **Undo -** Scan the log backward and undo the actions of the active transactions in the reverse order.

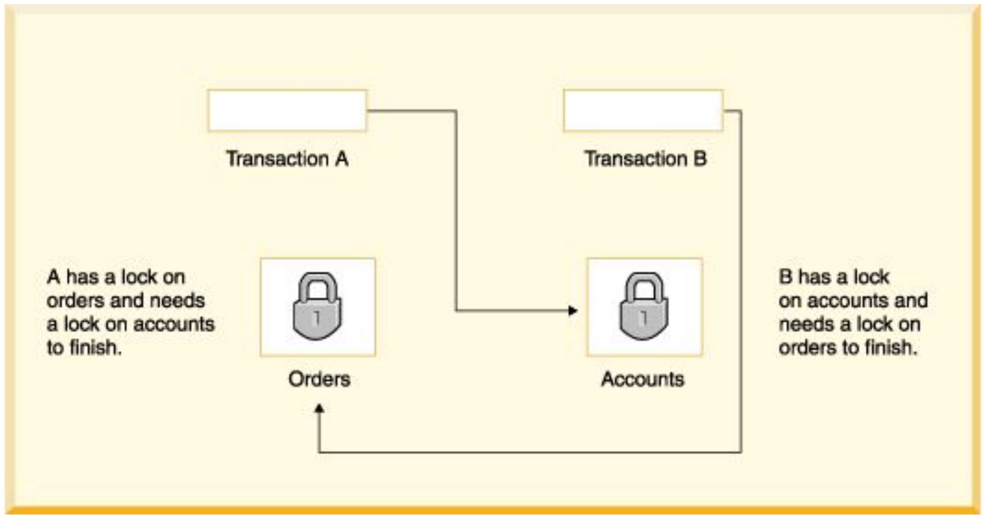
Next we will learn Deadlocks

Deadlocks occur when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T’ in the set.

Illustrating Deadlock Problem

Transaction A	Transaction B
Read_lock(orders)	
Read_item(orders)	
	Read_lock(accounts)
	Read_item(accounts)
Write_lock(accounts) (wait for accounts)	
	Write_lock(oders) (wait for oders)

Deadlock between Transaction A and B


Transaction A
Transaction B
A has a lock on orders and needs a lock on accounts to finish.
Orders
Accounts
B has a lock on accounts and needs a lock on orders to finish.

Approaches for dealing with deadlocks

1.Deadlock Prevention Protocols
  - 1.1. Based on Timestamp
    - 1.1.1. Wait-Die
    - 1.1.2. Wound-Wait
  - 1.2 Without using timestamp
    - 1.2.1. No-waiting
    - 1.2.2. Cautious waiting

2. Deadlock Detection Protocols
  - 2.1. Using wait-for graph

3. Timeouts

Deadlock Prevention Protocols

Based on Timestamp: 1. Wait-Die 2.Wound-Wait

Suppose Ti tries to lock an item X but is not able because X is locked by some other transaction Tj.

1. wait-die:
   - If TS(Ti) < TS(Tj), then (Ti older than Tj), Ti is allowed to wait
   - Otherwise (Ti younger than Tj) abort & rollback Ti and restart it using the same timestamp

4:00pm, Older Transaction **To**	4:05pm, Younger transaction **Ty**
Then **To** waits	If **Ty** holds X
If **To** holds X	Then **Ty** is discarded (die)

2. wound-wait:
   - If TS(Ti) < TS(Tj), then (Ti older than Tj), abort & rollback Tj, and restart it using the same timestamp
   - Otherwise (Ti younger than Tj), Ti is allowed to wait

4:00pm, Older Transaction **To**	4:05pm, Younger transaction **Ty**
If **To** holds X	Then **Ty** waits
Then **To** pre-empts (wounds) **Ty**	If **Ty** holds X

Deadlock Prevention protocol

Without using Timestamp

1. No-Waiting: If transaction cannot get lock it assumes deadlock. So it dies, waits and restarts later

2. Cautious Waiting

If transaction cannot get lock, it check the lock holder

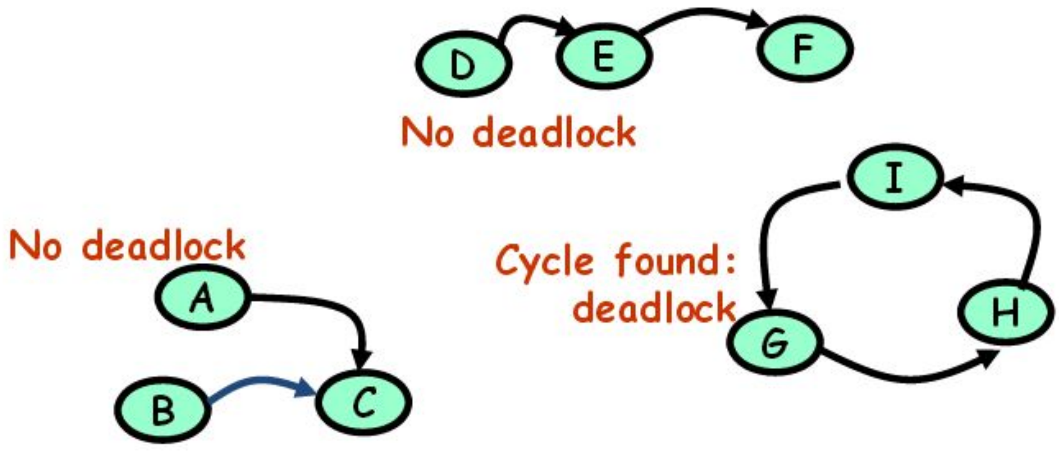
- If lock holder is already waiting then transaction dies and restarts later
- If Lock holder is active then transaction waits

Deadlock Prevention protocol

- **Timeouts**: If a transaction waits for a period longer than a **system-defined timeout period**, the system assumes that the transaction may deadlocked and aborts it – regardless of whether a deadlock actually exists or not

Deadlock Detection protocols

- In this approach, deadlocks are allowed to happen. The scheduler maintains a **wait-for-graph** for detecting cycle. **If a cycle exists**, then one transaction involved in the cycle is selected (victim) and rolled-back.
- A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle. One of the transaction of the cycle is selected and rolled back.



Starvation

- Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority based scheduling mechanisms.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.
- The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid starvation problem.

Approaches for dealing with deadlocks

1.Deadlock Prevention Protocols
    1.1. Based on Timestamp
        1.1.1. Wait-Die
        1.1.2. Wound-Wait
    1.2 Without using timestamp
        1.2.1. No-waiting
        1.2.2. Cautious waiting

2. Deadlock Detection Protocols
    2.1. Using wait-for graph

3. Timeouts

Thanks for Listening