Module 5: Transactions, Concurrency and Recovery, Recent Topics

PREPARED BY SHARIKA T R, SNGCE

PREPARED BY SHARIKA T R, SNGCE

SYLLABUS

- Transaction Processing Concepts - overview of concurrency control, Transaction Model, Significance of concurrency Control & Recovery, Transaction States, System Log, Desirable Properties of transactions. Serial schedules, Concurrent and Serializable Schedules, Conflict equivalence and conflict serializability, Recoverable and cascade-less schedules, Locking, Two-phase locking and its variations.
- Log-based recovery, Deferred database modification, check-pointing.
- Introduction to NoSQL Databases, Main characteristics of Key-value DB (examples from: Redis), Document DB (examples from: MongoDB)
- Main characteristics of Column - Family DB (examples from: Cassandra) and Graph DB (examples from : ArangoDB)

Single-User versus Multiuser Systems

- **Single-User System**:
  - At most one user at a time can use the system.
- **Multiuser System**:
  - Many users can access the system concurrently.
- **Concurrency**
  - **Interleaved processing**:
    - Concurrent execution of processes is interleaved in a single CPU
  - **Parallel processing**:
    - Processes are concurrently executed in multiple CPUs.

PREPARED BY SHARIKA T R, SNGCE

Transaction

- It is a group of database commands that can change/ access the data stored in a database
- Eg: Accounts

Id	Name	Balance
1	Mark	1000
2	Mary	1000

// Transfer Rs. 100 from Mark's Account to Mary's account

BEGIN TRANSACTION
Update Account Set Balance=Balance-100 where Id=1;
Update Account Set Balance=Balance+100 where Id=2
END TRANSACTION

- A transaction is treated as a single unit of work
- either all statements are succeeded or none of them
- Transaction maintains integrity of data in a database
- Eg, Transfer Rs. 100 from A to B:
- **Transaction boundaries**:
  - Begin and End transaction.

BEGIN TRANSACTION
R(A); //Read the balance of A
A=A-100;
W(A); //Write the updated value
R(B);
B=B+100;
W(B)
END TRANSACTION

TERMINOLOGY

- BEGIN TRANSACTION
  - Begining of transaction execution
- END TRANSACTION
  - Transaction execution is completed
- ROLLBACK_TRANSACTION(ABORT)
  - This signals that the transaction has ended unsuccessfully, so that any changes made by that transaction must be undone
- COMMIT_TRANSACTION
  - Signals successful end of the transaction. All changes made by the transaction are recorded permanently in the database and will not be undone

- A single **application program** may contain several transactions separated by the Begin and End transaction boundaries.
- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a read-only transaction; otherwise it is known as a read-write transaction.

SIMPLE MODEL OF A DATABASE FOR TRANSACTION PROCESSING

- **A database** is a collection of named data items
- **Granularity** of data
  - the size of a data item
  - a field, a record , or a whole disk block (Concepts are independent of granularity)
- Basic database access operations are **read** and **write**
  - **read_item(X)**: Reads a database item named X into a program variable also named X.
  - **write_item(X)**: Writes the value of program variable X into the database item named X.
- Basic unit of data transfer from the disk to the computer main memory is one block. 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

- 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.

PREPARED BY SHARIKA T R, SNGCE

write_item(X) command

- **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).
- the decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in coop_x0002_eration with the underlying operating system.

(a)

T_1

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

(b)

T_2

read_item(X);
$X := X + M$;
write_item(X);

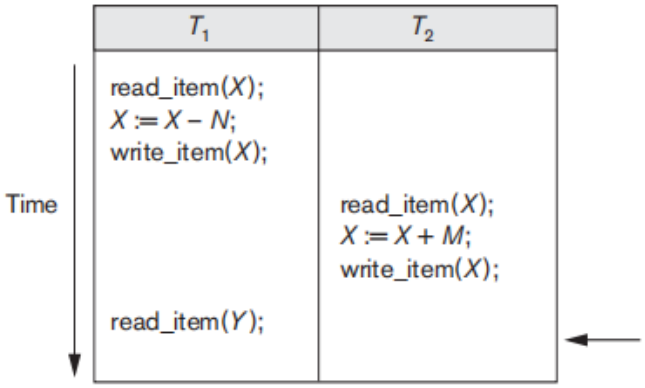
Figure 21.2
Two sample transactions. (a) Transaction T_1. (b) Transaction T_2.

Why Concurrency Control is needed

- The 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.

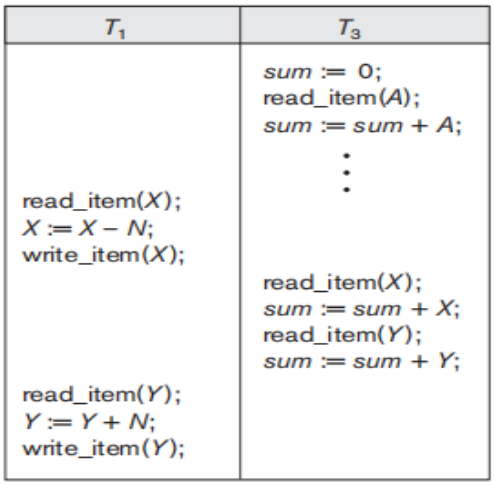
Time	T_1	T_2	
↓	read_item(X); $X := X - N$;		
		read_item(X); $X := X + M$;	
	write_item(X); read_item(Y);		
		write_item(X);	← Item X has an incorrect value because its update by T_1 is *lost* (overwritten).
	$Y := Y + N$; write_item(Y);		

- The Temporary Update (or Dirty Read) 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.



Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

- 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.



T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

- The Unrepeatable Read Problem
  - where a transaction T reads the same item twice and the item is changed by another transaction T' between the two reads.
  - Hence, T receives different values for its two reads of the same item.
  - This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights.
  - When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

Why recovery is needed

- 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 or tapes by mistake, and mounting of a wrong tape by the operator.

States of a Transaction

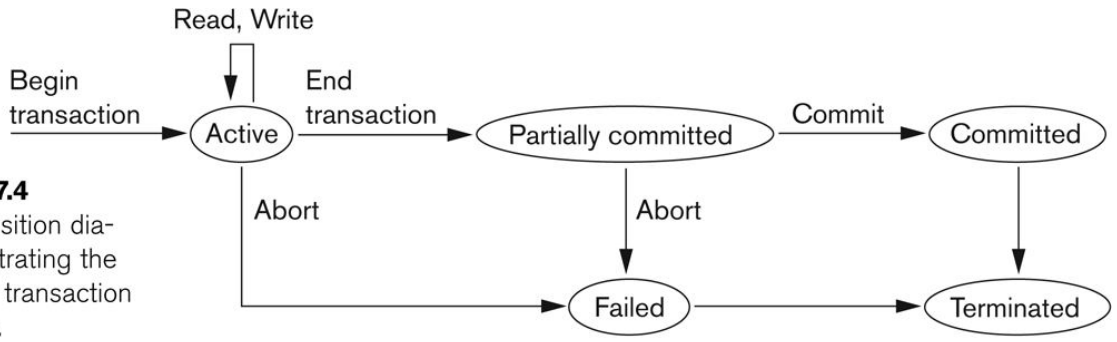


Figure 17.4
State transition diagram illustrating the states for transaction execution.

- Active
  - Statements inside the transaction block are executed
- Partially Committed
  - All updates/changes made by the transaction is applied on the data block availabel in the Main Memory Buffer. It is not yet updated in actual data block(Secondary Storage)
- Committed
  - Whenever the changes made by the transaction are permanently recorded in secondary storage , then we can say that the transaction is committed state. Now the changes cant be undone
- Terminated
  - Corresponds to transaction leaving the system
- Failed
  - Active to Failed: Some Commands are not able to complete
  - Partially Committed to Failed: Failure happens before making permanent changes

- Recovery manager keeps track of the following operations:
  - **begin_transaction**: This marks the beginning of transaction execution.
  - **read or write:** These specify read or write operations on the database items that are executed as part of a transaction.
  - **end_transaction**: This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
    - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.
  - **commit_transaction:** This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
  - **rollback (or abort):** This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

- Recovery techniques use the following operators:
  - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

- The System Log
  - **Log or Journal**:
    - ❖The log keeps track of all transaction operations that affect the values of database items.
      - This information may be needed to permit recovery from transaction failures.
      - The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
      - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

- T in the following discussion refers to a unique transaction-id that is generated automatically by the system and is used to identify each transaction:
- Types of log record:
  - [start_transaction,T]:
    - ➢Records that transaction T has started execution.
  - [write_item,T,X,old_value,new_value]:
    - ➢Records that transaction T has changed the value of database item X from old_value to new_value.
  - [read_item,T,X]:
    - ➢Records that transaction T has read the value of database item X.
  - [commit,T]:
    - ➢Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
  - [abort,T]:
    - ➢ Records that transaction T has been aborted.

- Protocols for recovery that *avoid cascading rollbacks do not require that read operations be written to the system log*, whereas other protocols require these entries for recovery.
- Strict protocols require simpler write entries that do not include new_value

Recovery using log records

- If the system crashes, we can recover to a consistent database state by examining the log
  - Because the log contains a record of every write operation that changes the value of some database item, it is possible to undo the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
  - We can also redo the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Commit Point of a Transaction

- A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log.
- Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.
- The transaction then writes an entry [commit,T] into the log.

Roll Back of transactions

- Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

Redoing transactions

- Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be redone from the log entries. (Notice that the log file must be kept on disk.
- At the time of a system crash, only the log entries that have been written back to disk are considered in the recovery process because the contents of main memory may be lost.)

PREPARED BY SHARIKA T R, SNGCE

Force writing a log

- Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk.
- This process is called force-writing the log file before committing a transaction.

Desirable Properties of a Transaction
ACID properties:

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

PREPARED BY SHARIKA T R, SNGCE

Atomicity

- A transaction is said to be atomic if either all of the commands are succeeded or none of them
- Responsibility of Recovery Subsystem
- Eg; Transfer Rs. 100 from A to B:

BEGIN TRANSACTION
R(A);
A=A-100;
W(A);
R(B);
B=B+100;
W(B)
END TRANSACTION

Either execute all command or no command at all

Consistency

- A transaction should be consistency preserving
- It should take the database from one consistent sate to another
- Reponsibility of Programers who wrote the database programs
- Eg: Transfer Rs. 100 from A to B

A=1000	→	A=900
B=1000		B=1100
Total=2000		Total=2000

PREPARED BY SHARIKA T R, SNGCE

Isolation

- Transactions should be isolate to each other during concurrent execution
- Responsibility: Cocurrent control subsystem

Durability or Permanency

- The changes applied to the database by committed transaction must persist in the database
- these changes must not be lost because of any failure
- Responsibility of Recovery Subsystem

PREPARED BY SHARIKA T R, SNGCE

Schedules(Histories) of Transactions

- A schedule S of n transactions T1, T2,.. Tn is an ordering of the operation of the transactions
- When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).

1. Serial Schedule
2. Non Serial Schedule
3. Serializable Schedule

Serial Schedule

PREPARED BY SHARIKA T R, SNGCE

- Entire transactions are performed in serial order
- Only One transaction is active at a time
- Serial schedule : **T1T2**

T1	T2
R(A)	
A=A+50	
R(B)	
B=B-30	
W(B)	
	R(A)
	A=A+40
	W(A)
	R(B)
	B=B-60
	W(B)

	Initially	Final Value
A	100	190
B	200	110

Serial Schedule

			A	B	C
	Read (A, t)		500	500	500
	t = t - 100				
T1	Write (A, t)				
	Read (B, t)				
	t = t + 100				
	Write (B, t)		400	600	500
		Read (A, s)			
		s = s - 100			
		Write (A, s)			
T2		Read (C, s)			
		s = s + 100			
		Write (C, s)	300	600	600

300 + 600 + 600 = 1500

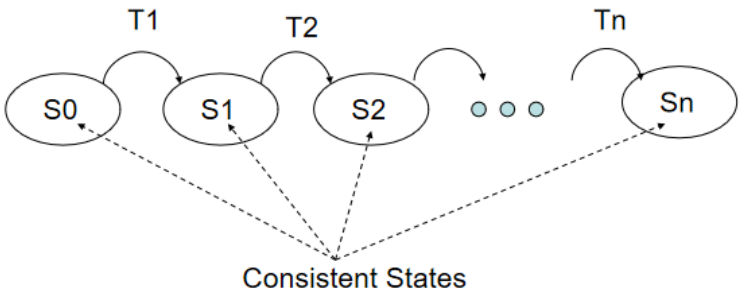
Serial Schedule

			A	B	C
		Read (A, s)	500	500	500
		s = s - 100			
T2		Write (A, s)			
		Read (C, s)			
		s = s + 100			
		Write (C, s)	400	500	600
	Read (A, t)				
	t = t - 100				
T1	Write (A, t)				
	Read (B, t)				
	t = t + 100				
	Write (B, t)		300	600	600

300 + 600 + 600 = 1500

PREPARED BY SHARIKA T R, SNGCE

- Every serial schedule is considered as correct if the transactions are independent
- it does not matter which transaction is executed first
- T1 T2 T3...Tn =T2T3T4T1... TN=T3Tn T1 T4 T6...T10
- All these serial schedules are correct schedules



Problems with serial schedules

1. Poor resource utilization
2. More waiting time
3. Less throughput
4. Late response

Serial schedules are unacceptable in practice even though they are correct

Non Serial Schedules

- Executing the transaction in an interleaved or instructions of transactions are interleaved. Schedule S1

T1	T2
R(A)	
A=A+50	
W(A)	
	R(A)
	A=A+40
	W(A)
R(B)	
B=B-30	
W(B)	
	R(B)
	B=B-60
	W(B)

Initially	Final Value
A=100	A=190
B=200	B=110

Schedule S1 gives the correct result same as serial schedule

Schedule S2

T1	T2
R(A)	
A=A+50	
	R(A)
	A=A+40
	W(A)
W(A)	
R(B)	
B=B-30	
	R(B)
	B=B-60
	W(B)
W(B)	

Initially	Final Value
A=100	A=150
B=200	B=170

Schedule S2 gives the erroneous result
not same as serial schedule

Summary

- Serial Schedules always gives correct result
- But there are some drawbacks
- In order to avoid that we introduced non serial schedules
- But not all non serial schedules are correct
- So we need a non serial schedule which always give a correct result
- or Equivalent to a Serial Schedule = Serializable Schedule

Serializable Schedule

- A non serial schedule of n transactions is said to be serializable if it is equivalent to some serial schedule of same n transactions
- Every serializable schedules are considered as correct

PREPARED BY SHARIKA T R, SNGCE

When are two schedules considered equivalent?

- Result equivalence
- Conflict equivalence
- View equivalence

Result equivalence

- Two schedules are said to be result equivalent if they produce the same final state of the database they produce same result

S1	S2

read_item(X);
X:=X+10;
write_item(X);
	read_item(X);
	X:=X*1.1;
	write_item(X);

For X=100: S1 & S2 are result equivalent
But for other values, not result equivalent
//S1 and S2 should not be considered equivalent

PREPARED BY SHARIKA T R, SNGCE

Conflict equivalence

- Conflict Operations
  - Two operations in a schedule are said to conflict if they satisfy
    1. They belong to different transactions
    2. They access the same data item
    3. At least one of the operation is a write operation

- Eg: Schedule S:

R1(X)	W2(X)	: Conflict
R2(X)	W1(X)	:Conflict
W1(X)	W2(X)	:Conflict
R1(X)	R2(X)	:No Conflict
W2(X)	W1(Y)	:No Conflict
R1(X)	W1(X)	:No Conflict

Why conflicting: Read Write Conflict

X=10;		
R1(X)	W2(X)	//X=20
W2(X)//20	R1(X)	
//T1 reads 10	//T1 reads 20	

Write Write Conflict

X=10;	
W1(X) //20	W2(X) //30
W2(X) //30	W1(X) //20
//Final Value 30	//Final Value 20

PREPARED BY SHARIKA T R, SNGCE

Read Read Non Conflicting

X=10;	
R1(X)	R2(X)
R2(X)	R1(X)
//T1 & T2 reads 10	//T1 & T2 reads 10

Conflict Equivalence

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

Check whether the following two schedules are conflict equivalent or not.

Schedule S1:

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

Schedule S2:

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

S1 and S2 are conflict equivalent

Check whether the following two schedules are conflict equivalent or not.

Schedule S1:

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

Schedule S2:

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

No conflicting operations in S1 and S2 .
So they are conflict equivalent

Check whether the following two schedules are conflict equivalent or not.

Schedule S1:

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

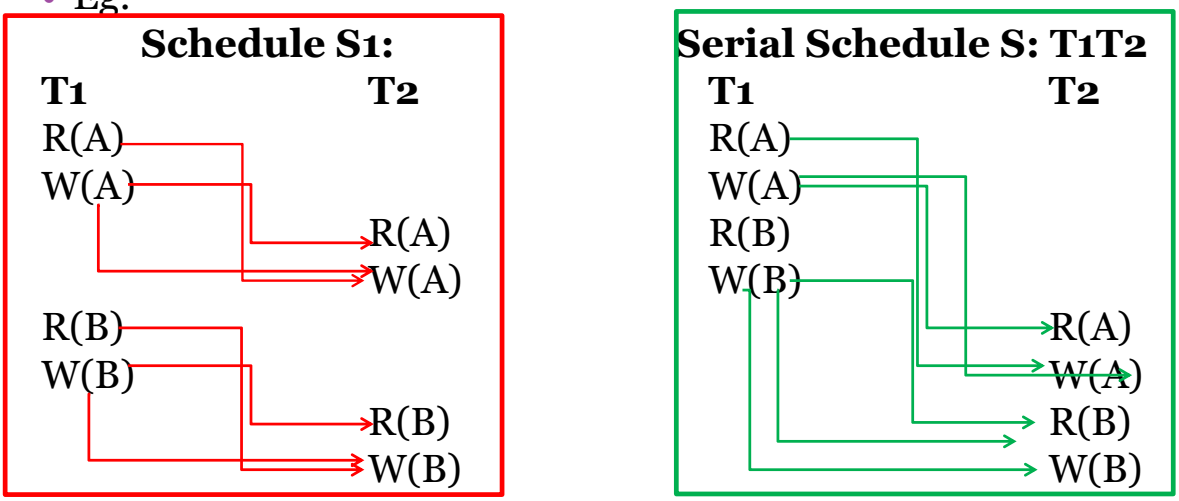
Schedule S2:

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

S1 and S2 are **not** conflict equivalent

Conflict Serializable Schedules

- If a non serial schedule is conflict equivalent to some serial schedule then it is called Conflict serializable schedules
- Eg:



S1 and S are conflict equivalent. So S1 is a conflict serializable schedule.
We can also say that S1 is serializable

PREPARED BY SHARIKA T R, SNGCE

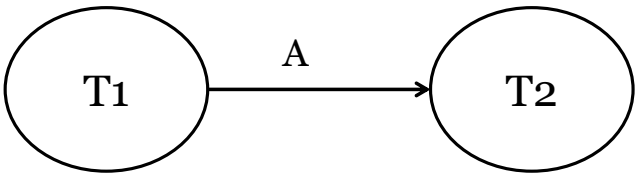
- To check a schedule is conflict serializable or not, we can use precedence graph

If the precedence graph is free from cycles(acyclic) then schedule is conflict serializable schedule

Rules for constructing Precedence Graph:
Rule 1

S1:

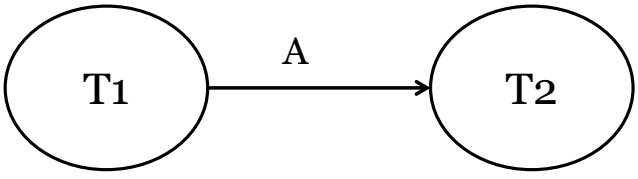
T1	**T2**
R(A)	
	W(A)



Rule 2

S2:

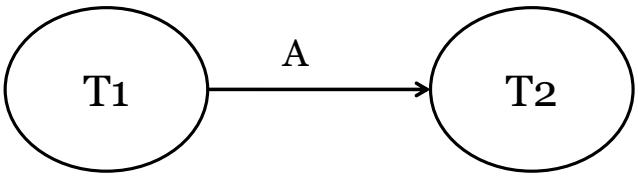
T1	**T2**
W(A)	
	W(A)



Rule 3

S3:

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



Rule 4

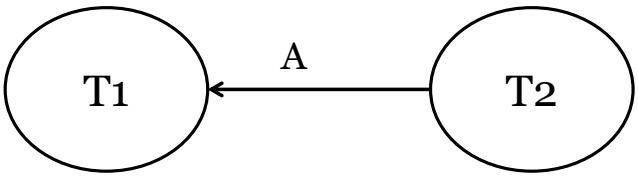
S4:

T1	**T2**
R(A)	
	R(A)

Rule 5

S5:

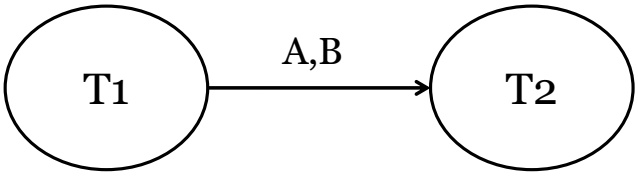
T1	T2
	W(A)
R(A)	



Check if S1 is conflict serializable

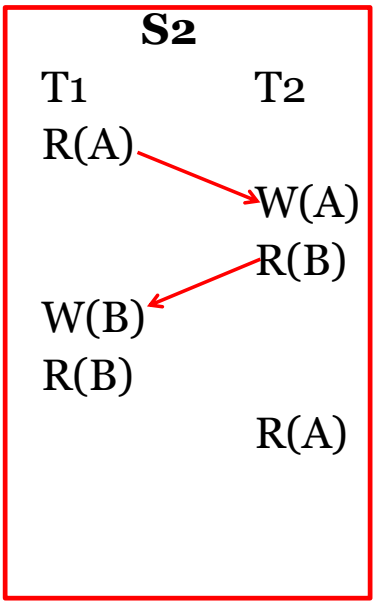
S1

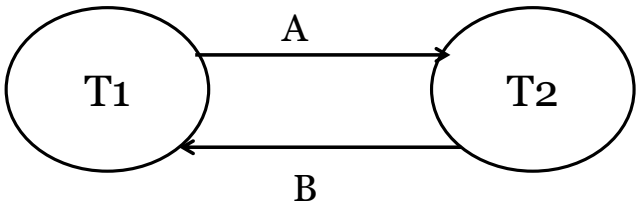
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)



No Cycle in precedence graph so S1 is conflict serializable

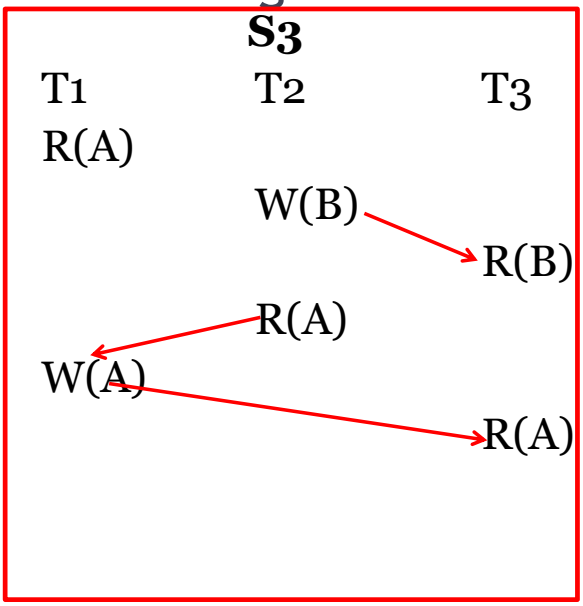
Check if S2 is conflict serializable

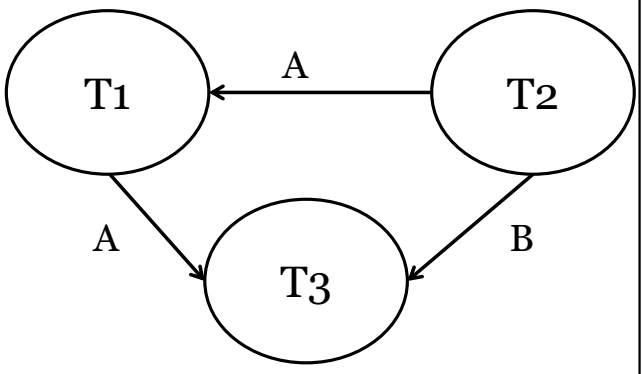




Cycle in precedence graph so S1 is NOT conflict serializable

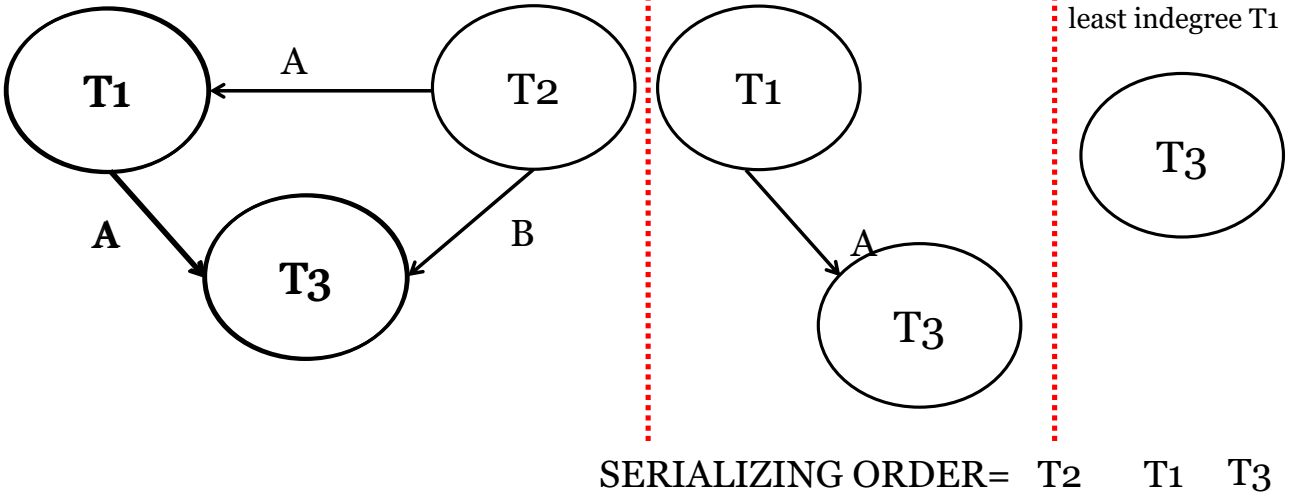
Check if S3 is conflict serializable and find its serializing order





NO Cycle in precedence graph so S1 is conflict serializable.

Serializing order



SERIALIZING ORDER= T2 T1 T3

View Equivalence

- Two schedules S1 and S2 are said to be view equivalent if for each data item,
  - If the initial read in S1 is done by Ti, then same transaction should be done in S2 also
  - If the final write in S1 is done by Ti, then in S2 also Ti should perform the final write
  - If Ti reads the item written by Tj in S1, then S2 also Ti should read the item written by Tj

View Serializable Schedule

- A non serial schedule is view serializable schedule if it is view equivalent to some of its serial schedule

Check if S1 is view serializable

S1

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

S2: T1T2

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

S1 and S2 is view serializable hence serializable

Initial read in S1 by T 1 and S2 T2
Final write in S1 is by T2 AND S2 is by T2
Writer reader conflict in data item is same sequence in both S1 AND s2

Important Points

- Every Conflict serializable schedule is View Serializable but vice versa is not true
- Every Conflict Serializable schedule is Serializable but vice versa is not true
- Every View Serializable schedule is serializable and vice versa is also true

PREPARED BY SHARIKA T R,
SNGCE

Shortcut to check View Serializable or not

Step 1: All conflict Serializable schedule are View Serializable

Step 2: For those schedules that are not conflict serializable

If there does not exist any blind writes, then the schedule is surely not view serializable

// Blind Write: Performs a write operation without reading

S1

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

Not Conflict Serializable and No Blind Write=Not View Serializable

If there exist blind write, then the schedule may or may not be view serializable. The follow Normal Procedure

Characterizing Schedule Based on Recoverability

- Once a transaction is committed, we can't abort or rollback
- If a transaction is failed, it can't commit

1. Irrecoverable and Recoverable Schedule
2. Cascadeless Schedule
3. Strict Schedule

Irrecoverable Schedule

- If a transaction Tj reads a data item written by transaction Ti and commit operation of T j is before the commit of Ti, then such schedules are irecoverable

S

T1	T2	
R(A)		
A=A-20		//A=-10
W(A)		
	R(A)	//A=-10
	Commit;	// A=-10 is permanent
R(B)		
W(B)		
Commit;		

Initially A=10
Here T2 is not recoverable ie irrecoverable
T2 performs a dirty read

if T1 fails here t1 should roolback A to 10 but A's value is already committed. It cannot rollback. A=-10 is an invalid value

Recoverable Schedules

- If a transaction Tj reads a data item written by transaction Ti, then the commit operation of T j should happen only after the commit Ti, then such schedules are recoverable
- Tht is Writer_transaction should commit first, then Reader_transaction commit (Recoverable Schedules)

S

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

//A=10

First T1 has first write so it must commit here it is so.

So T2 is recoverable in case if there is a failure in T1

S

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

Irrecoverable

S

T1	T2
R(A)	
W(A)	
Commit;	
	W(A) //Blind Write
	Commit;

Recoverable

S

T1	**T2**	**T3**
R(A)		
W(A)		
	R(A)	
	W(A)	
		R(A)
		W(A)
C1;		
	C2	
		C3

Recoverable

The problem with above schedule is **cascading rollback/aborts**

Cascading Rollbacks

- Because of a single failure in a particular transaction, if all dependent transactions need to be rolled back, it is called as cascading rollbacks
- Such schedules are recoverable, but can be time consuming since numerous transaction csn be rolled back
- Complete waste of work, time and resources
- To avoid cascading rollbacks, we need Cascadeless schedules

PREPARED BY SHARIKA T R, SNGCE

Cascadeless Schedules

- Schedule which are free from Cascading Rollbacks
- If every transaction in the schedule reads only items that were written by committed transaction, then such a schedule is called Cascadeless schedules

	S	
T1	**T2**	**T3**
R(A)		
W(A)		
C1;		
	R(A)	
	W(A)	
	C2	
		R(A)
		W(A)
		C3

Recoverable and Cascadeless

Every cascadeless schedules are Recoverable but vice versa is not true

	S
T1	**T2**
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
	C;
C;	

Recoverable NOT Cascadeless

Cascading rollbacks/ abort possible

Strict Schedule

- More restrictive type of schedule
- In a schedule, a transaction is neither allowed to read/write a data item until the transaction that has write is committed or aborted

S	
T1	**T2**
W(A)	
	W(A)
C;	

Recoverable, Cascadeless, Not Strict

PREPARED BY SHARIKA T R, SNGCE

S	
T1	**T2**
W(A)	
C	
	R(A)/W(A)
	C;

Recoverable, Cascadeless, Strict

If a schedule is strict, then it is Recoverable as well as Cascadeless. But vice versa is not true

Concurrency Problems

- When multiple transactions are running concurrently in an uncontrolled or unrestricted manner in a schedule, then it might lead to several problems

1. Dirty Read Problem
2. Unrepeatable Read Problem
3. Lost Update Problem
4. Phantom Read Problem

PREPARED BY SHARIKA T R, SNGCE

Dirty Read Problem

- A transaction is reading a data written by an uncommitted transaction is called as dirty read

S

T1	**T2**
R(A)	
W(A)	
	R(A) //Dirty read
	C;

Failed

- T2 reads the dirty value of A witten by uncommitted transaction T1
- T1 fails in later stages and rollbacks
- Thus the value that T2 read now stand to be incorrect
- Therefore database become inconsistent

Example

Id	ProductName	ItemInStock
1	iPhone	1

S

T1	**T2**
Update ItemInStock=0 (Billing Customer)	
	Reads ItemsInStock=0 //dirty read C;
Transaction failed because of Insufficient funds and rollback	

Lost Update Problem

- It happens when two or more transactions read and update the same data item

S

T1	**T2**
R(A) //50	
A=A-10 //40	
	R(A) //50
	A=A-20 //30
W(A)	
	W(A)

Here the update made by T1 is lost. T2 overwritten the update of T1. Because of lost update we are getting 30 instead of 20

Example

Id	ProductName	ItemInStock
1	iPhone	1

S

T1	**T2**
Read ItemInStock=10	
	Reads ItemsInStock=10
Sell 1 item	*Sell 2 items(ItemInStock-=2)*
	Update ItemsInStock=8
Update ItemsInStock=9	

Here the update made by T2 is lost. T1 overwritten the update of T2. Because of lost update we are getting 9 instead of 7

Unrepeatable Read Problem

- It happens when one transaction reads the same data twice and another transaction updates that in data between the two reads.

S

T1	**T2**
R(A) //10	
	R(A) //10
	A=A-20 //-10
	W(A)
R(A)//-10	

Here T1 is not able to repeat the same read. T1 Wonders how the value of Y got changed because according to it, it is running in isolation

Example

Id	ProductName	ItemInStock
1	iPhone	1

S

T1	T2
Read ItemInStock //10	
	Update ItemsInStock=8
Doing some other work	
Read ItemsInStock //8 →	Unrepeatable read problem

Phantom Read Problem(Incorrect Summary Problem)

- Case 1:
  - It happens when one transaction executes a quert twice and it gets a different number of rows in result set each time
  - This happens when a 2nd transaction inserts a new row that matches the where clause of the query executed by 1st transaction

Id	Name
1	Mark
3	Mary
100	Tony

S

T1	T2
Select * from Employee where Id between 1 and 3 // 2 rows	
Doing some other work	Insert into Employee values(2,"John");
select * from Employee where Id between 1 and 3 //3 rows	// Extra tuple is called a Phantom Tuple

Here T2 wonders who deleted the variable X because according to it, it is running in isolation

- To ensure consistency of the database we have to prevent the occurence of the above problems
- CONCURRENCY CONTROL PROTOCOLS helps to prevent the occurence of above problems and maintainthe consistency of the database

CONCURRENCY CONTROL PROTOCOLS

- Serial schedules are corrct but they are unacceptable
- We can have a number of no serial schedules
- We need a non serial schedule which is equivalent to some of the serial schedule ----> Serializable Schedule
- To check whether a given schedule is serilizable or not, we can use conflict equivalence test and view equivalence test
- In order to acheive serializability and to prevent the occurence of concurrency problems in non serial schedules we are using the concurrency control protocols

If every individual transaction follows concurrency control protocol then all schedules in which those transaction participate become serializable

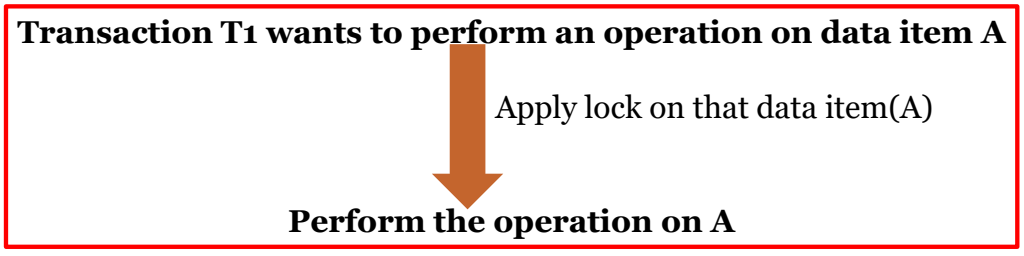
Three type of Concurrency Control Protocols

1. **Lock based Protocols**
2. Time stamp based Protocol
3. Graph based protocols

PREPARED BY SHARIKA T R, SNGCE

Lock Based Protocols

- Locking protocol means, whenever a transaction wants to perform any operation(Read/Write) on any data item, first it should request for lock on that data item.
- Once it acquire the lock then only it can perform the operation
- Locking mechanism is managed by Concurrency Control Subsystem



Types of Lock

- Two type of Locks are

1. Shared Lock(S(A))
2. Exclusive Lock(X(A))

PREPARED BY SHARIKA T R, SNGCE

Shared Lock

- If a transaction wants to perform read operation on a data item, it should apply shared lock on that item

T1
S(A)
Read(A)

Exclusive Lock

- If a transaction acquire this lock, it can perform read/write/both operation on that data item

T1
X(A)
Read(A)
Write(A)

PREPARED BY SHARIKA T R, SNGCE

Lock Compatibility Matrix

- Concurrency control manager manages the lock based on Lock CompatibiLity matrix

		Tj	
		S(A)	**X(A)**
Ti	**S(A)**	Yes	No
	X(A)	No	No

Case 1

- Multiple transactions can perform read operation on same data item at the same time
- More thane one shared locks are possible

T1	T2 . . .	...	Tn
S(A)			
R(A)			
	S(A)		
	R(A)		

Case 2

- If any transaction want to apply an exclusive lock on data item on which already a shared lock is allowed for some other transaction is Not Possible

T1	T2	Tn
S(A)		
R(A)		
	S(A)	
	R(A)	
		X(A)//Not allowed
		W(A)

Case 2... cont..

- But T3 can aquire this lock after unlocking

T1	T2	Tn
S(A)		
R(A)		
	S(A)	
	R(A)	
U(A)		
	U(A)	
		X(A)//allowed
		W(A)

Multiple shared locks on same data item is allowed
But all other combinations are not allowed

Qn 1. Whether the following is lock compatible or not

T1	T2
S(A)	
R(A)	
S(B)	
R(B)	
U(A)	
	X(A)
	R(A)
	X(B)
	R(B)

B is not unlocked.
so this exclusive lock is not possible

So **NOT Lock Compatible**

Qn 2. Whether the following is lock compatible or not

T1	T2
S(A)	
R(A)	
U(A)	
S(B)	
R(B)	
	X(A)
	W(A)
	S(B)
	R(B)
	U(A)
	U(B)
U(B)	

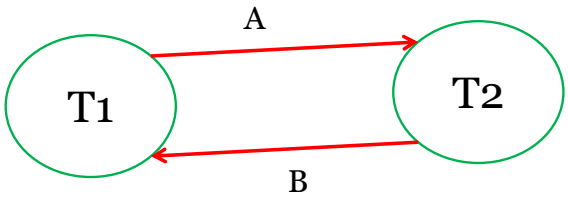
So **Lock Compatible**

This is called as **Simple Locking Protocol**

Qn 2. Given a non serial schedule

T1	T2
S(A)	
R(A)	
U(A)	
	X(A)
	W(A)
	U(A)
	X(B)
	W(B)
	U(B)
S(B)	
R(B)	
U(B)	

So **Lock Compatible,**
But it is **not Serializable**



Problems with Simple Locking Protocol

I. Sometimes does not guarantee Serializability
II. May lead to deadlock

To guarantee serializability, we must follow some additional protocol concerning the *positioning of locking and unlocking* operation in every transaction

Two phase Locking (2PL)

- If every individual transactions follows 2PL, then all schedules in which these transactions participate become Conflict Serializable
- 2PL Schedule are always Conflict Serializable Schedules

PREPARED BY SHARIKA T R, SNGCE

- A schedule is said to be in 2PL if all transactions perform locking and unlocking in 2 phases

1. Growing Phase:
   - New locks on data items can be acquired but none can be unlocked
   - Only locking is allowed no unlocking
2. Shrinking Phase
   - Existing locks may be released but no new locks can be acqired
   - Only unlocking is allowed no locking

T1

Lock Apply (Growing Phase)
//perform operations
Lock Release(Shrinking Phase)

Important Points

- Every transaction should first perform Growing phase and then Shrinking phase
- After Shrinking phase, Growing phase is not possible
- Growing phase is compulsory, but shrinking phase is optional
- 2PL schedules are always Conflict Serializable but vice versa is not true
- **Lock point:** The point at which a transaction acquire the last lock

Qn 1. Whether the following is 2PL or not

T1	T2
X(A)	
R(A)	
W(A)	
S(B)//LP	
R(B)	
	S(B)//LP
	R(B)
	U(B)
U(B)	
U(A)	

So **Lock Compatible, 2PL, Serializable(T1T2)**

Qn 2. Whether the following is 2PL or not

T1	T2
S(A)	
R(A)	
	S(B)//LP
	R(B)
	U(B)
X(B)	
R(B)	
W(B)	
U(B)	
	S(B)
	R(B)
	U(B)

So **Lock Compatible, not in 2PL,**

Problem with Basic 2PL

1. Basic 2PL does not ensure Recoverability
2. Cascading rollbacks are possible
3. Deadlocks are possible

Basic 2PL does not ensure Recoverability

T1	T2
S(B)	
X(A)//LP	
R(B)	
	S(B)
	R(B)
R(A)	
W(A)	
U(A)	
	X(A)//LP
	R(A)
	W(A)
	U(A)
	C;
C;	

First Write so first commit should be first done by T1 but T2 comit first, so this is irrecoverable

Lock Compatible, 2PL, Irrecoverable

Cascading rollbacks are possible

T1	T2	T3
X(A)//LP		
R(A)		
W(A)		
U(A)		
	X(A)//LP	
	R(A)	
	W(A)	
	U(A)	
C;		
		S(A)//LP
		R(A)
	C;	
		C;

Lock Compatible, 2PL, Recoverable NOT Cascadeless

Deadlocks are possible

T1	T2
X(A)	X(B)
R(A)	R(B)
W(A)	W(B)
**********	**********
Wants to update B	Wants to update A
**********	**********

Here none of the transaction can continue. None of them will release the locks
Because once they release any locks then it is not possible to lock any data item
This will leads to deadlock

- These drawback can be removed by using the following variation of basic 2PL
- We have to solve Recoverability, Cascading Rollbacks and Deadlock

Variations of 2PL

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

PREPARED BY SHARIKA T R, SNGCE

Strict 2PL

- Follow basic 2PL + All exclusive locks should unlock only after commit operation
- Strict Schedule
  - A transaction is neither allowed to read/write a data item until the transaction that has written is committed

T1	T2
W(A)	
C;	
	R(A)/W(A) //Strict Schedule

Benifits and Drawbacks of Strict 2PL

- Benifits
  - Generates Conflict Serializable scchedules(variation of Basic 2PL)
  - Produce Strict Schedules (hence it is recoverable and cascadeless)
- Drawback
  - Deadlock is possible

Example

T1	T2
X(A)	
R(A)	
S(B)	
R(B)	
	S(B)
	R(B)
W(A)	
U(B)	
C;	
U(A)	
	X(A)
	R(A)
	W(A);
	C;

Exclusive lock on A unlocked after comit(unlock not mandatory)

Lock Compatible, 2PL, Strict 2PL

Rigorous 2PL

- More stronger than Strict 2PL
- Follow basic 2PL + All locks (both exclusive and shared locks) should unlock only after commit operation
- Every Rigorous 2PLis Strict 2PL but vice versa is not true

PREPARED BY SHARIKA T R, SNGCE

Benifits and Drawbacks of Rigorous 2PL (Same as Strict 2PL)

- Benifits
  - Generates Conflict Serializable scchedules(variation of Basic 2PL)
  - Produce Strict Schedules (hence it is recoverable and cascadeless)
- Drawback
  - Deadlock is possible

Example

T1	T2
S(A)	
R(A)	
	S(B)
	R(B)
	X(C)
	R(C)
	W(A)
	C
	U(B);
	U(C)
S(C)	
R(C)	
C;	

Rigorous 2PL & Strict 2PL

Conservative 2PL(C2PL)

- No DEADLOCK
- Follow Basic 2PL + All transaction should obtain all lock they need before the transaction begins
- Release all lock after commit
- Recoverable, Serializable, Casadeless, Deadlock Free
- Not practical to implement

Steps for Conservative 2PL(C2PL)

Step 1: All Locks should acquire at the beginning(before transactionexecution begins

Step 2: Perform operations(Read/Write)

Step 3: On completion release all locks

C2PL is free from deadlock: A transaction will begin its execution only if all locks are available so there is no chance of waiting for any resources during execution time (No Hold and Wait)

Example

T1(wants A and B)		**T2(wants A and B)**
X(A)	//Step 1	
X(B)		
R(A)	//Step 2	
R(B)		
W(A)		
W(B)		
C		
U(A);	//Step 3	
U(B)		
		T2 CAN START

Drawbacks of Conservative 2PL

- Poor resourse utilization
- Concurrency is limited
- Each transaction needs to declare all the data items that need to be read/write at beginning, which is not always possible

PREPARED BY SHARIKA T R, SNGCE

Log-based recovery

- A **Log** is the most widely used structure for recording database modifications.
- **Update log record**: It describes a single database write. It has following fields-
  - **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
  - **Data item** identifier is the unique identifier of the data item written.
  - **Old value** is the value of the data item prior to the write.
  - **New value** is the value of the data item that it will have after the write.

- Various types of log records are represented as:
- **< Ti start>:** Transaction Ti has started.
- **<Ti, X, V1, V2> :** Transaction Ti has performed a write on data item Xj , Xj had value v1 before the write, and will have value v2 after the write.
- **<Ti commit> :** Transaction Ti has committed.
- **<Ti abort> :** Transaction Ti has aborted.
- Two techniques that users log:
  - Deferred database modification
  - Immediate database modification

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 <Ti start> record to log.
- A write(X) operation results in a log record <Ti, X, V> being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When Ti partially commits, <Ti commit> 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 <Ti start> and<Ti commit> are there in the log.
- Redoing a transaction Ti (redo Ti) 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

Example Transaction To and T1 (To execute before T1)

example transactions T_0 and T_1 (T_0 executes before T_1):

T_0	T_1
read (A)	**read** (C);
A: = A – 50;	C:-=C- 100;
Write (A);	**write** (C);
read (B);	
B:= B + 50;	
write (B);	

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo(To) must be performed since <To commit> is present

(c) redo(To) must be performed followed by redo(T1) since <To commit> and <Ti commit> are present

Immediate Database Modification

- The immediate database modification technique allows database modifications to be output to the database while the transaction is still in the active state.
- 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 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 Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
T_0, B, 2000, 2050		
	$A = 950$	
	x_1 $B = 2050$	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	$C = 600$	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.

Checkpointing Protocol

- All committed transactions in the log file before checkpoint are permanently saved on the disk. So no need to anything
- All committed tansactions after checkpoint should be redone (redo list)
- All uncommitted transactions (before and after checkpoint) should be undone (undo list)

- 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 record <Ti start>, but does not contain the record <Ti commit>.
  - Transaction Ti needs to be redone if the log contains both the record <Ti start> and the record <Ti commit>.
- Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

Recovery With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.
  - All transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume concurrency control using strict two-phase locking;
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
- Logging is done as described earlier.
  - Log records of different transactions may be interspersed in the log.
- The checkpointing technique and actions taken on recovery have to be changed
  - since several transactions may be active when a checkpoint is performed.

- Checkpoints are performed as before, except that the checkpoint log record is now of the form
  < **checkpoint** *L*>
  where *L* is the list of transactions active at the time of the checkpoint
  - We assume no updates are in progress while the checkpoint is carried out (will relax this later)
- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found.
     For each record found during the backward scan:
     - if the record is <T_i **commit**>, add T_i to *redo-list*
     - if the record is <T_i **start**>, then if T_i is not in *redo-list*, add T_i to *undo-list*
  3. For every T_i in *L*, if T_i is not in *redo-list*, add T_i to *undo-list*

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  1. Scan log backwards from most recent record, stopping when <T_i **start**> records have been encountered for every T_i in *undo-list*.
     - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
  2. Locate the most recent <**checkpoint** *L*> record.
  3. Scan log forwards from the <**checkpoint** *L*> record till the end of the log.
     - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

<T_0 **start**>
<T_0, *A*, 0, 10>
<T_0 **commit**>
<T_1 **start**>
<T_1, *B*, 0, 10>
<T_2 **start**> /* Scan in Step 4 stops here */
<T_2, *C*, 0, 10>
<T_2, *C*, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 **start**>
<T_3, *A*, 10, 20>
<T_3, *D*, 0, 10>
<T_3 **commit**>