View Serializability

- Offers less restrictive definition of schedule equivalence than conflict serializability.
- Two schedules are view serializable, If the following rules are followed while creating the second schedule out of the first.

View Serializability

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.
2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.
3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

View Serializability

- The idea behind view equivalence is that, 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.
- The read operations are hence said to see the same view in both schedules.
- Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules.

View Serializability

- Schedule is view serializable if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is view serializable, although converse is not true.
- It can be shown that any view serializable schedule that is not conflict serializable contains one or more blind writes.

Example - View Serializable Schedule

- T1: r1(X); w1(X); T2: w2(X); and T3: w3(X);

S1 : r1(X); w2(X); w1(X); w3(X);

w2(X) and w3(X) – blind writes

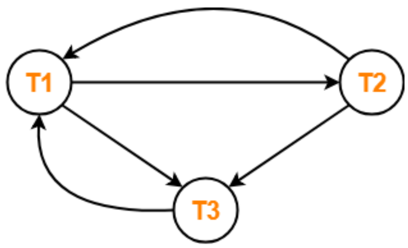
Schedule S1 is view serializable since it is equivalent to the serial schedule T1, T2, T3.

Example - View Serializable Schedule

- R1(A) , W2(A) , R3(A) , W1(A) , W3(A);

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

$R_1(A)$, $W_2(A)$ ($T_1 \rightarrow T_2$)
$R_1(A)$, $W_3(A)$ ($T_1 \rightarrow T_3$)
$W_2(A)$, $R_3(A)$ ($T_2 \rightarrow T_3$)
$W_2(A)$, $W_1(A)$ ($T_2 \rightarrow T_1$)
$W_2(A)$, $W_3(A)$ ($T_2 \rightarrow T_3$)
$R_3(A)$, $W_1(A)$ ($T_3 \rightarrow T_1$)
$W_1(A)$, $W_3(A)$ ($T_1 \rightarrow T_3$)

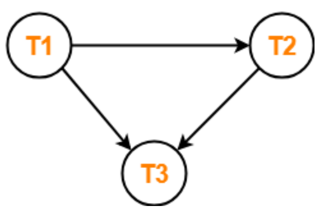


- There exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.

Example - View Serializable Schedule

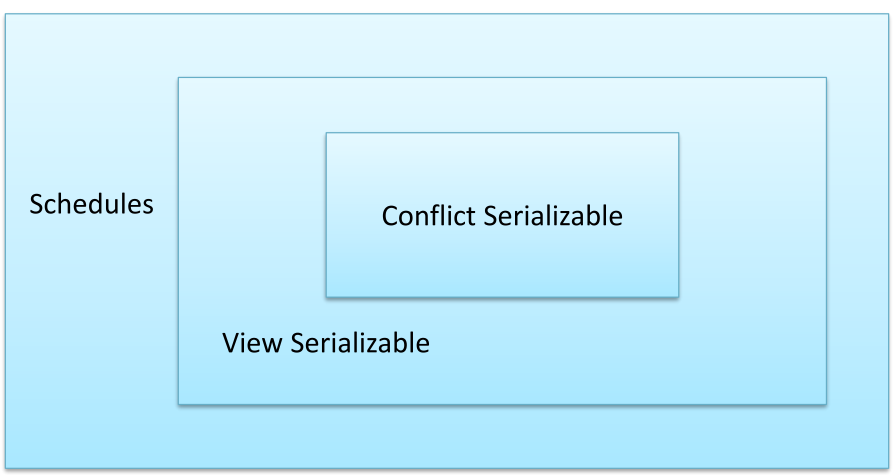
Dependency Graph

- T1 first reads A and T2 first updates A. So, T1 must execute before T2 and thus the dependency **T1 → T2**.
- Final update on A is made by the transaction T3. So, T3 must execute after all other transactions. Thus the dependency **(T1, T2) → T3**.
- From write-read sequence, the dependency **T2 → T3.**



- Clearly, there exists no cycle in the dependency graph.
- Therefore, the given schedule S is view serializable.
- The serialization order T1 → T2 → T3.

Summary



Schedules Based on Recoverability

- Schedules can be characterized according to the following terms:
  - (1) recoverability,
  - (2) avoidance of cascading rollback, and
  - (3) strictness.
- Those properties of schedules show successively more stringent conditions.

Schedules Based on Recoverability

- Schedules in which transactions commit only after all transactions whose changes they read commit.
- The schedules that theoretically meet this criterion are called *recoverable schedules.*
- Those that do not are called non-recoverable, and hence should not be permitted.

MASTER OF COMPUTER SCIENCE/
MASTER OF SCIENCE IN COMPUTER SCIENCE

Concurrency Control Techniques

Dr. Jeevani Goonetillake

UNIVERSITY OF COLOMBO SCHOOL OF COMPUTING

Concurrency Control

Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

Concurrency Control

- Concurrency-control protocols are defined to allow concurrent schedules and at the same time to make sure that the schedules are conflict/view serializable, and are recoverable and maybe even cascadeless.
- These protocols enforce a mechanism that avoids non-seralizable schedules instead of testing a schedule for Serializability after it has executed through examining the precedence graph.

Concurrency Control Protocol Types

- **Lock Based Protocol**
- **Time-Stamp Ordering Protocol**
- Multi-version Protocol
- Graph Based Protocol
- Multiple Granularity Protocol

Concurrency Control

LOCKS

- To control concurrent execution of transactions locking of the data items technique is used.
- A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operation can be applied to it.
- Generally, there is one lock for each data item in the database.
- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Database Concurrency Control

- Lock Manager:
  - Managing locks on data items.
- Lock table:
  - Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Lock Types

Different Types are available

- Binary
- Shared/exclusive

Binary Locks

- A binary lock can have two states: locked and unlocked.
- A distinct lock is associated with each database item X.
- If
  - Lock(X)= 1→ item X cannot be accessed by a database operation that requests the item.
  - Lock(X)=0→ Can access and change the value to 1
  - Unlock_item(X) will release the lock
- Two operations, lock_item and unlock_item, are used with binary locking.
- Enforces Mutual Exclusion on the data item.

Binary Lock Operations

Lock_item(X)

if LOCK (X) = 0 (*item is unlocked*)

then LOCK (X) ← 1 (*lock the item*)

else begin

wait (until lock (X) = 0) and

the lock manager wakes up the transaction);

goto B

end;

Binary Lock Operations

Unlock_item(X)

LOCK (X) ← 0 (*unlock the item*)

if any transactions are waiting then

wake up one of the waiting the transactions;

Rules for Binary Locks

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.

2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.

3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.

4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

Shared/Exclusive (or Read/Write) Locks

- Binary locking scheme is too restrictive because at most one transaction can hold a lock on a given item.
- Should allow several transactions to access the same item X if they all access X for *reading purposes only.*
- Multiple-mode lock is used and there are two locks modes:
  - (a) shared (read)
  - (b) exclusive (write)

Shared/Exclusive (or Read/Write) Locks

- **Shared mode: shared lock (X)**

  More than one transaction can apply share lock on X for reading its value, but no write lock can be applied on X by any other transaction.

- **Exclusive mode: Write lock (X)**

  Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Shared/Exclusive (or Read/Write) Lock Operations

- The following code performs the read operation:

```
B: if LOCK (X) = "unlocked" then
    begin LOCK (X) ← "read-locked";
    no_of_reads (X) ← 1;
  end
 else if LOCK (X) ← "read-locked" then
     no_of_reads (X) ← no_of_reads (X) +1
  else
        begin
          wait (until LOCK (X) = "unlocked" and
          the lock manager wakes up the transaction);
          go to B
end;
```

Shared/Exclusive (or Read/Write) locks

The following code performs the write lock operation:

```
B: if LOCK (X) = “unlocked”
   then LOCK (X) ← “write-locked”;
   else begin
     wait (until LOCK (X) = “unlocked” and
      the lock manager wakes up the transaction);
    go to B
  end;
```


Shared/Exclusive (or Read/Write) Lock Operations

The following code performs the unlock operation:

```
if LOCK (X) = “write-locked” then
  begin LOCK (X) ← “unlocked”;
     wakes up one of the transactions, if any
  end
  else if LOCK (X) ← “read-locked” then
    begin
       no_of_reads (X) ← no_of_reads (X) -1
       if  no_of_reads (X) = 0 then
        begin
           LOCK (X) = “unlocked”;
          wake up one of the transactions, if any
       end
    end;
```

Shared/Exclusive Lock Rules

- A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
- A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
- A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
- A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.

Shared/Exclusive Lock Rules

- A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may be relaxed.
- A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Example Problem

Initial values: X=20, Y=30

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

- If they are executed as two serial schedules T1, T2 or T2, T1 then serializability is guaranteed.

Example Problem

Time

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

Concurrency will be serializable only if it gives the result of one of the serial schedules.