IN3020/4020 – Database Systems
Spring 2021, Week 8.2

Serialization and Concurrency Control
Part 2

Dr. M. Naci Akkøk, Chief Architect, Oracle Nordics

Based upon slides by E. Thorstensen from Spring 2019

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Another type of serialization protocol

- So far, we have only looked at 2PL-based protocols. All 2PL protocols have the following structure:
  - First, we lock the data elements we are interested in
  - Then we read and (possibly) write these data elements
  - In the end we release the locks and give access to other transactions
- Such protocols are well suited when the data is stored in tabular form (such as arrays and hash tables)
- The other main way to store data is to organize them as trees (usually B trees or B + trees)
- For such data there is a more convenient locking protocol, called the tree protocol

The tree protocol

1. A transaction can put its first lock on an arbitrary node in the tree
2. Later, the transaction can only lock a node if it has locked the parent node
3. A transaction can delete a lock it has on a node at any time
4. A transaction cannot lock a node that it has previously released (even though it still has a lock on the parent node)

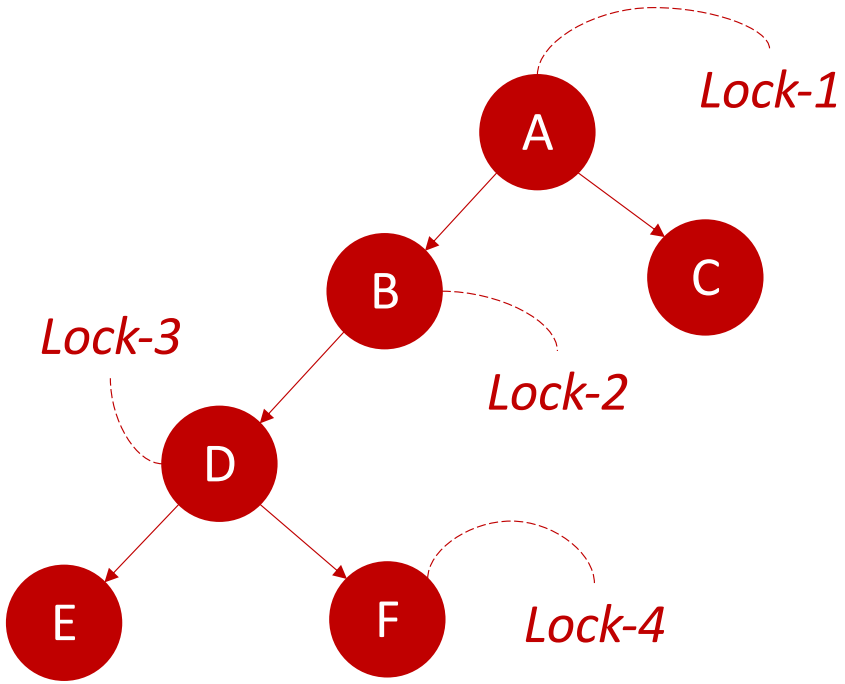
The tree protocol formulated here is based upon having only one type of lock, but it works equally well with several.

Tree protocol on B + trees

- In a B + tree, the root is always locked first
- In theory, the transaction must hold a lock on the root until the transaction is complete because both insert and delete can cause the root to be changed
- It would have meant that only one writing transaction at a time could access the B + tree
- However, the normal thing is that when you access the next level in the tree, you can immediately determine that the parent node will not be affected, and it can then be released immediately
- In practice, this gives a high degree of simultaneity

An example of the tree protocol

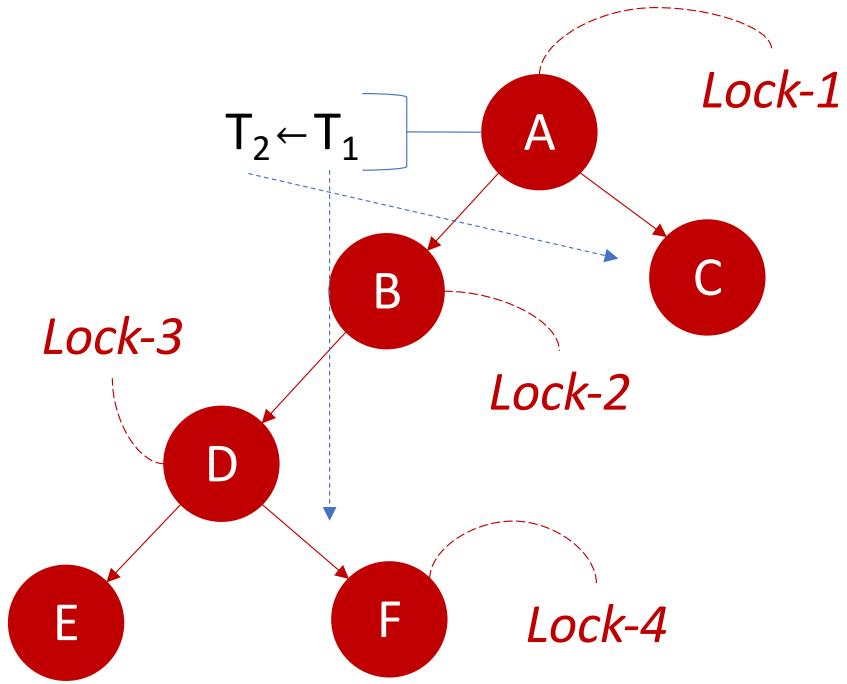
1. T locks A (root, Lock-1)
2. T locks B (Lock-2)
3. T sees that A is not changed and removes Lock-1
4. T locks D (Lock-3)
5. T sees that B is not changed, and removes Lock-2
6. T locks F (Lock-4)
7. T sees that D will be changed
8. T writes F and D and removes Lock-3 and Lock-4



Extract of a B+ tree with root A

The tree protocol ensures serializability!

- Suppose that T_1 is as before, that T_2 will also write and that T_1 locks A first
- T_2 tries to lock A and is placed in the A queue until T_1 releases Lock-1
- T_2 gets lock A and can move on to C, but not to B until T_1 has released lock-2
- T_2 can never "pass" T_1, so the order at the root determines a serialization order!



Extract of a B+ tree with root A

Time stamping

- Time stamping is the basis for a family of serialization protocols that do not use locks
- Timestamp provides optimized concurrency control where the transactions can go unimpeded until you discover that something went wrong such the transaction must be aborted (Remember: 2PL is pessimistic and tries to prevent mistakes in advance)
- When a transaction T starts, it receives a **timestamp**, TS(T)
- Two transactions cannot have the same timestamp, and if T_1 starts before T_2, then we should have $TS(T_1) < TS(T_2)$
- The serialization order is determined by the timestamps
- The two most common forms of timestamps are:
  - The time T started (the value of the system clock)
  - A serial number (transaction number in the system´s life-time)

Time stamps on data

What does the scheduler schedule?
EXECUTION plans!

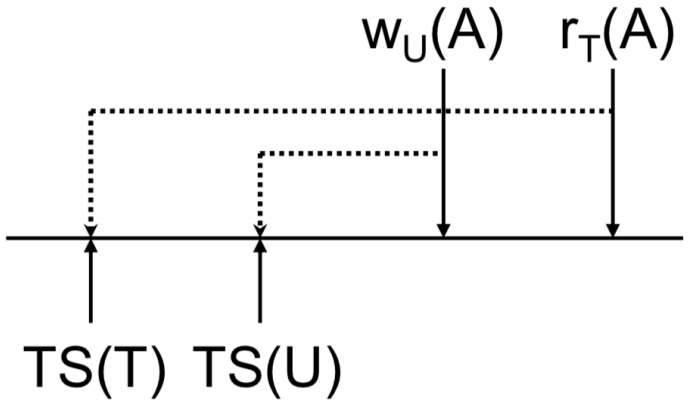
- The Scheduler must maintain a table of active transactions (as before) and their timestamps now
- To able to use timestamp for concurrency checking, two timestamps and a Boolean variable are associated with each and every data item A in the database:
  - RT(A): **read time of A**: The highest timestamp of any transaction that has read A
  - WT(A): **write time of A**: The highest timestamp of any transaction that has written A
  - C(A): **commit flag of A**: True if and only if the last transaction that wrote A, is committed

Time stamp serialization

- The serialization protocol using timestamps basically assumes that the serialization order is determined by the timestamps
- The protocol ensures that we get an execution plan that is conflict equivalent to the serial plan we would have had if all the transactions had done all their read and write operations at their timestamp (that is, if the transactions were instantaneous and did not take any time from start to finish)
- There are two problems that can occur:
  - The transaction reads too late
  - The transaction writes too late

Transaction reads too late

- Transaction T will read data element A
- The value of A is written by a transaction that started after T, i.e., WT(A) > TS(T)
  - T would read «wrong» value of A
  - Consequence: T must abort

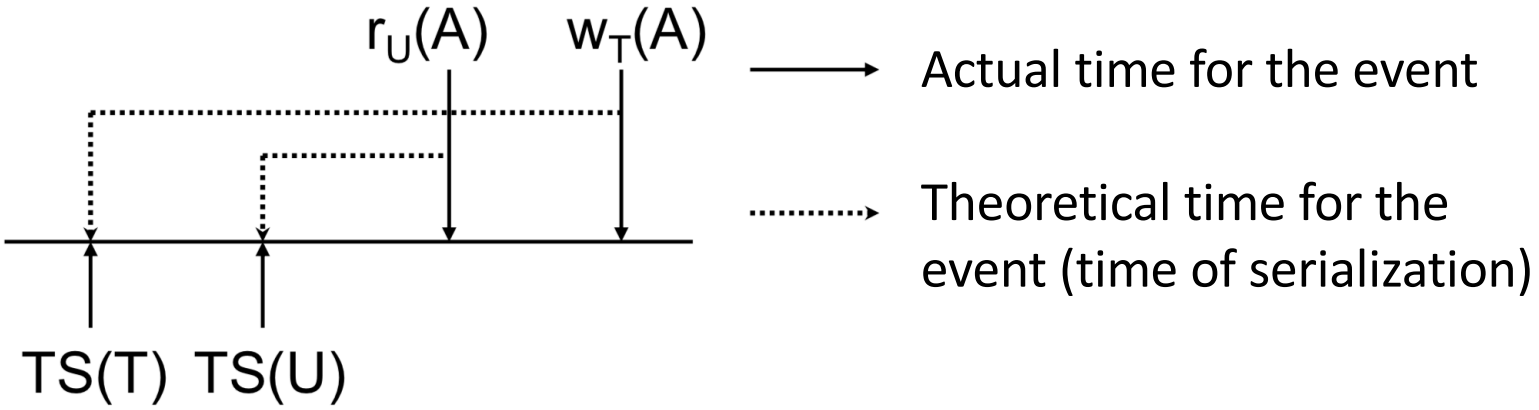


Actual time for the event

Theoretical time for the event (time of serialization)

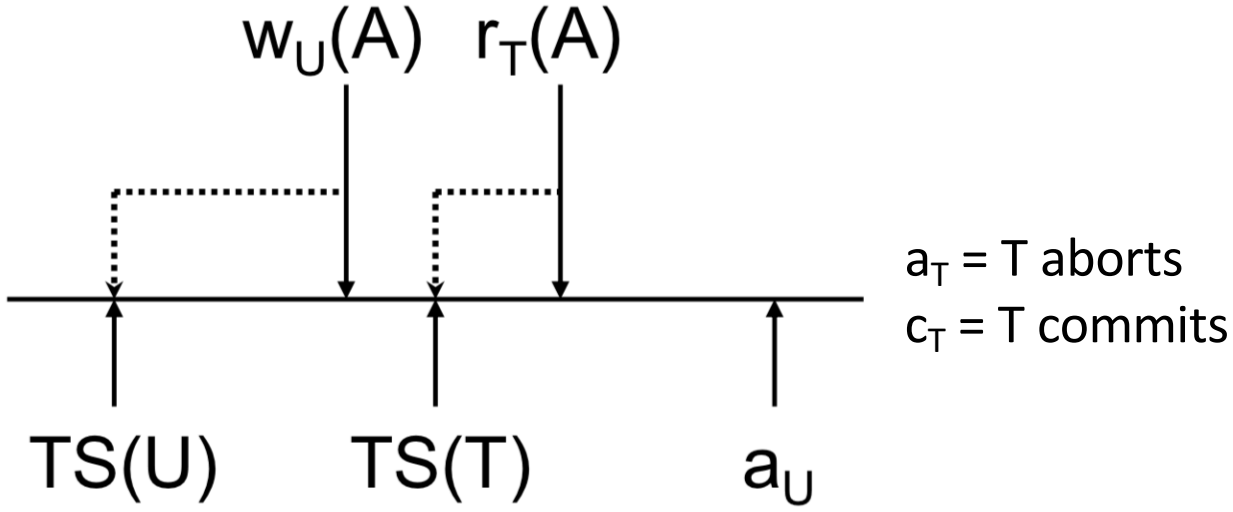
Transaction writes too late

- T wants to write data element A, but A is already read by U that started after T, i.e., RT(A) > TS(T)
- If WT(A) > TS(T), T shall not write A (everything is OK)If not, U should have read the value of A that T should now write, and T must then abort



Dirty data

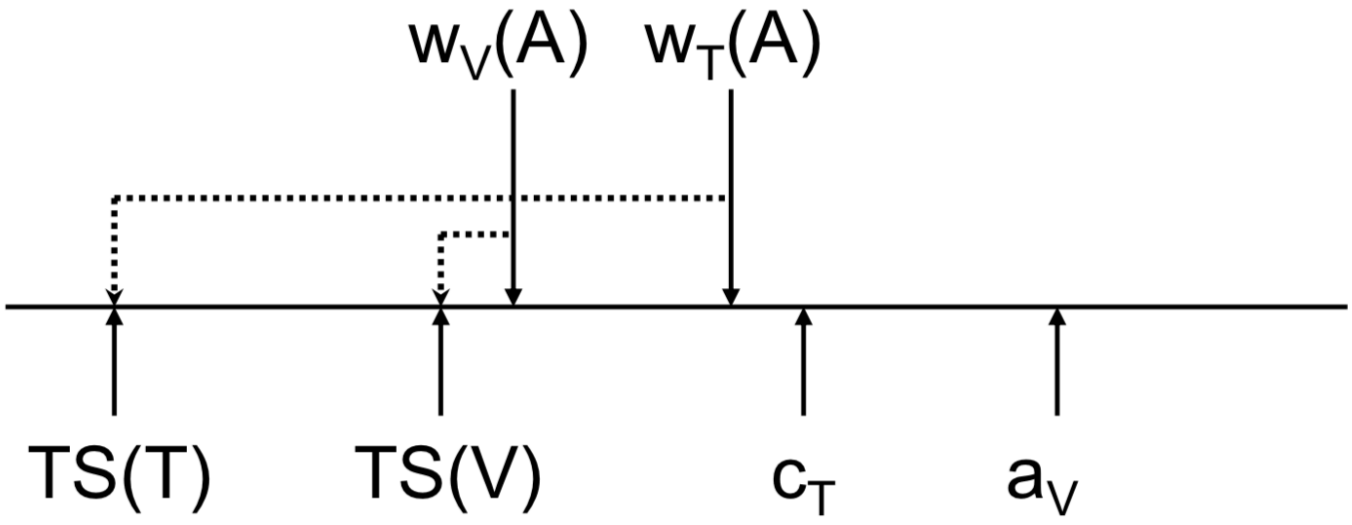
- Transaction T reads data element A written by another transaction U
- If U aborts after T has read A, T has read a value of A that should never have been in the database! We say that T has read a dirty value of A
- This is a violation of the isolation requirement, so T should wait to read A until commit flag of A is true



Thomas´writing rule

- Transaction T wants to write data element A already written by another transaction V with TS(V) > TS(T)
- Since later transactions should read the A value written by V, T must not write A

Problem:
If V aborts, T should not have written A!



Timestamp serialization protocol I

1) T wants to read A
   a) If TS(T) ≥ WT(A), T may be allowed to read A:
      i. If C(A), i.e., if the commit flag of A is set, then perform $r_T(A)$ and set RT(A) = max (RT(A), TS(T))
      ii. If not C(A), let T wait in line for A (when T wakes up, it must test again)
   b) If TS(T) < WT(A), it is physically impossible for T to read the correct value of A. T must be rolled back (we must perform rollback(T)):
      i. Perform a_T
      ii. Restart with a higher timestamp

Timestamp serialization protocol II

2) T wants to write A
   a) If TS(T) ≥ RT(A), and TS(T) ≥ WT(A), T can write:
      i. Perform w_T and set WT(A) = TS(T) and C(A) = false
      ii. "Inform" those waiting for A
   b) If TS(T) ≥ RT(A) and TS(T) < WT(A), then A is written by a newer transaction than T. If C(A), ignore the request (Thomas' writing rule). If not C (A), let T wait in the A queue (and test again when T is awakened)
   c) If TS(T) < RT(A), A is read by a newer transaction than T, and T must be rolled back

Timestamp serialization protocol III

3) T wants to commit
   a) Execute C_T (i.e., enter commit(T) into the log)
   b) For each A where T was the last transaction that wrote A, set C(A) = true
   c) Allow transactions pending to read or write A to continue
4) T is aborted (or wants to abort)
   a) Use the log to perform Undo on all write operations of T and write abort(T) in the log
   b) Let all transactions that are waiting to read or write a data item written by T try again

Versioning

- Versioning is a method for avoiding abort in the case when T wants to read an A written by a newer transaction
  - When a write operation $w_T(A)$ is performed, a new version A_t of A is created, where t = TS(T)
  - When a read operation $r_T(A)$ is performed, it reads the version that has the highest $t \leq TS(T)$
  - Obsolete versions of A can, and should, be deleted
  - For A_t to be deleted, there must be a version A_u where $t < u$ and where $u \leq TS(T)$ for all active transactions T
  - Effective versioning in practice requires that the data elements are blocks

Snapshot Isolation (SI)

- Commercial DBMSs use a form of versioning called "Snapshot Isolation", abbreviated SI
- In SI, a transaction T throughout all its lifetime will read the committed values the database had at TS(T)
- T is thus not affected by writing operations of other transactions after TS(T)
- SI does not guarantee serializability (!)
- But SI provides a high degree of merging (interweaving) and is the standard strategy for simultaneous control in most of today's commercial DBMSs.

Time stamping vs. locking

- Time stamping is best if most transactions are read only, or if conflicts are rare
- If conflicts are common, timestamping will result in many rollbacks, and locking is better

- Take some time to discuss why…
- And why not both? How?

Validation

- Validation is an optimistic serialization strategy based on time stamping
- It differs from regular timestamps in that it does not store read and write timestamps for all data elements in the database
- For each active transaction T, two quantities (sets) are stored
  - the read set of T, RS(T)
  - the write set of T, WS(T)

  which contains all data elements that T reads or writes, respectively

Validation (continued)

The execution of a transaction T is divided into three stages:

1. **The reading phase**
   All reading and calculation is done here. RS(T) and WS(T) is built up in T's address space. The start time of the reading phase is called Start(T)
2. **The validation phase**
   T is validated by comparing RS(T) and WS(T) with the read and write sets of other transactions (details soon). If the validation fails, T is rolled back.
   The end time of the validation phase is called Val(T)
3. **The writing phase**
   Here, T writes the values in WS(T) to the database. The end time of the Writing phase is called Fin(T)

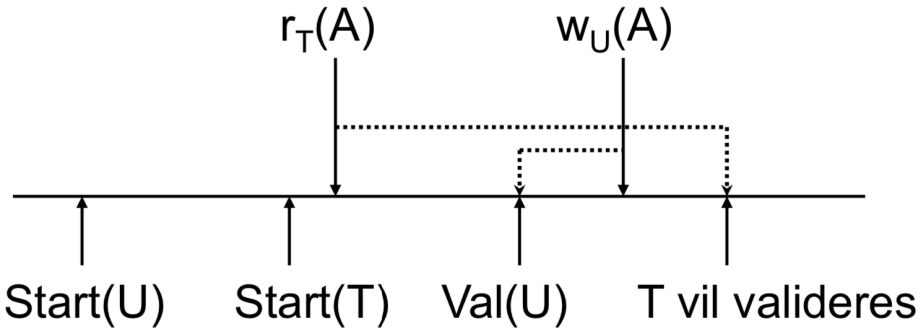
The value of Val(T) determines the serialization order

Validation (continued)

The planner (scheduler) maintains three sets of transactions:

1. START
   Those who have started but have not yet completed the validation phase.
   For each T ∈ START, Start(T) is stored
2. VAL
   Those that are validated but have not completed the writing phase.
   For each T ∈ VAL, Start(T) and Val(T) are stored
3. FIN
   Those who (recently) have completed the writing phase.
   For each T ∈ FIN, Start(T), Val(T) and Fin(T) are stored.
   T can be removed from FIN when for all U ∈ START ∪ VAL,
   we have Start(U) > Fin(T)

Validation phase – Example 1



Assume that when T is to be be validated, there is a U such that

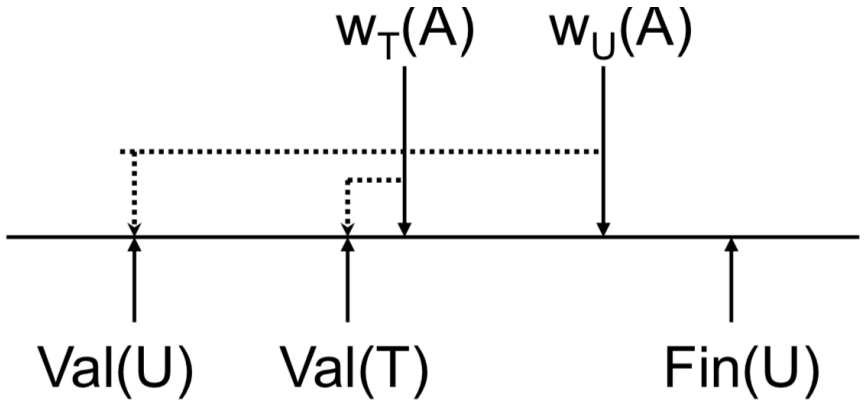
a) U ∈ VAL ∪ FIN (i.e., U is validated)

b) If U ∈ FIN: Fin(U) > Start(T) (i.e., U was not finished when T started)

c) RS(T) ∩ WS(U) ≠ Ø (In the figure, A ∈ RS(T) ∩ WS (U))

T must be rolled back because T may have read a "wrong" value of A

Validation phase – Example 2



Assume that when T is to be be validated, there is a U such that

a) U ∈ VAL (i.e., U is validated but not finished)

b) WS(T) ∩ WS(U) ≠ Ø (In the figure, A ∈ WS(T) ∩ WS(U))

T must be rolled back because T may write A before U does

The validation test

The two previous examples cover all possible «fault» situations.

- Thus, the validation phase consists of the following two tests:
  - Check that RS(T) ∩ WS(U) = Ø for all U ∈ VAL and all U ∈ FIN with Fin(U) > Start(T)
  - Check that WS(T) ∩ WS(U) = Ø for all U ∈ VAL
- If T passes both tests, T is validated and enters the writing phase
- If not, T must be rolled back

Cascaded rollback

When T_1 aborts, the scheduler deletes all the locks T_1 has.

If T_2 is allowed to continue, then T_2 will create an inconsistent state, so T_2 must be rolled back (because T_2 has read a dirty A)

T_1	T_2	Integrity rule A = B	A	B
			25	25
$l_1(A)$; $r_1(A)$; A←A+100;				
$w_1(A)$; $l_1(B)$; $u_1(A)$;			125	
	$l_2(A)$; $r_2(A)$;			
	A←Ax2; $w_2(A)$;		250	
	$l_2(B)$; *Rejected!*			
$r_1(B)$;				
a_1; $u_1(B)$;				
	$l_2(B)$; $u_2(A)$; $r_2(B)$;			
	B←Bx2; $w_2(B)$; $u_2(B)$;			50
			250	50

Cascaded rollback (continued)

- As the name suggests, cascaded rollback can be recursive:
  - Abort of T1 can lead to abort of T2, which then can lead to abort of T3, etc.
- Cascaded rollback can (without extra precautions) include committed transactions, which is in violation of D in ACID
  - Cascaded rollback of uncommitted transactions should also be avoided
- Timestamp protocols with commit flags protect against cascaded rollback
- Validation also protects against cascaded rollback (no writing is done until we know that there will be no abort)

We continue with concurrency…

We will be looking at

recoverable execution plans, and then **isolation**

Recoverable execution plans

- An execution plan is recoverable if no transaction T commits until all transactions that have written data that T has read have committed.
- Examples (hint: look at which transaction has read which data):
  - Example 1: A recoverable serializable (serial) plan:
    S_1: $w_1(A)$; $w_1(B)$; $w_2(A)$; $r_2(B)$; c_1; c_2
  - Example 2: A recoverable but not conflict-serializable plan:
    S_2: $w_2(A)$; $w_1(B)$; $w_1(A)$; $r_2(B)$; c_1; c_2
  - Example 3: A serializable but non-recoverable, plan:
    S_3: $w_1(A)$; $w_1(B)$; $w_2(A)$; $r_2(B)$; c_2; c_1

Recoverable execution plans (continued)

- In order for an execution plan to be recoverable, the commit records must be written to disk in the same order as they are written in the log (multiple log entries can be in the same block and be written at the same time)
- This applies to both undo, redo and undo/redo logging

ACR (Avoid Cascade Rollback) plans

- An execution plan avoids cascade rollbacks if the transactions can only read data written by committed transactions
- Such plans are called ACR (Avoid Cascade Rollback) plans

- All ACR plans are recoverable

  Proof: Suppose we have an ACR plan.
  Suppose that T_2 reads a value written by T_1 after T_1 has committed.
  Since T_2 did neither commit nor abort before it reads, T_2 must do its commit or abort after T_1 made its commit. Q.E.D.

Strict locking

- An execution plan uses strict locking if it is based on locks and adheres to the following rule:

  Strict lock rule: A transaction cannot release any write lock until it has committed or aborted, and the commit or abort log entry is written to disk

- An execution plan that uses two-phase locking and first releases the write locks after commit/abort is called strict 2PL. The reader locks can be released at any time during the shrinkage phase.

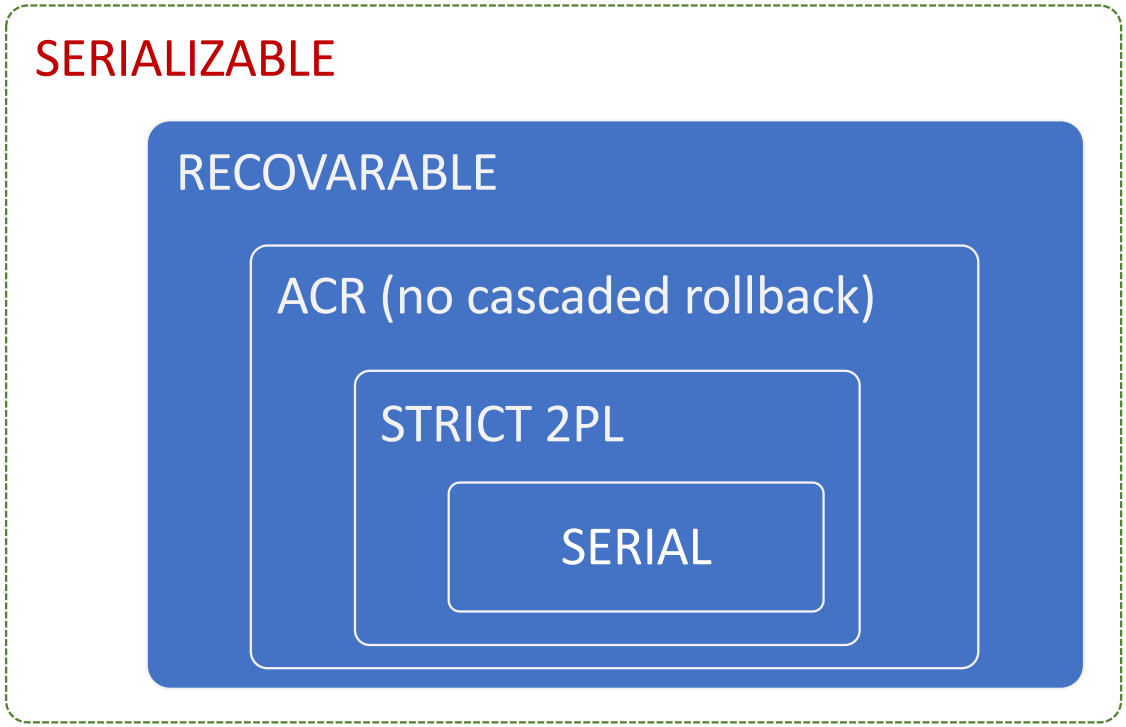
Strict locking (continued)

- An execution plan with strict locking is an ACR plan

  Proof: Because the write locks are not released until after the transactions commit, no one can read data written by a noncommited transaction. Q.E.D.

- Strict locking alone is not sufficient to ensure serializability. However, strict 2PL is serializable.

Execution plan types



Rolling back in case of locks

- If the data elements are blocks, everything is simple:
  - All writing is done in the buffer; nothing is written to disk before commit
  - In case of abort, the block is released, which becomes unused/available buffer area
  - The same technique works in versioning; the block with the «aborted version» is released
- If there are multiple data elements in each block, there are three ways to restore data after an abort
  1. The original can be read from the database on disk
  2. With an undo or undo/redo log, the original can be retrieved from the log
  3. Each active transaction can have its own log of its own changes in the memory

Group commit

- With group commit, locks can be released earlier than with the strict locking rule
- Group commit:
  - A transaction cannot release any write lock until it has committed or aborted, and the commit or abort log entry is written to (primary) memory
  - Log blocks must be flushed to the log disk in the order they are in the primary memory

Group commit (continued)

- **Group commit gives recoverable plans**
  - Proof: Suppose T_1 writes X and commits, and that T_2 reads X. T_2 can only read dirty data if the system crashes before T_1's commit record is on disk. But in that case, T_2's commit record can not be on the disk either, so both T_1 and T_2 are aborted by the Recovery Manager. Thus, this prevents dirty data from being read. Q.E.D.
- **Group commit gives serializable plans**
  - Proof: The plans are conflict equivalent with the serial plans we get by allowing each transaction to be executed at the time of the commit. Q.E.D.

Logical locking

- Logical log is a log-type that uses the transaction logic for the rollbacks
- Typical logical log entries consist of four fields
  - **L**: a log entry serial number
  - **T**: transaction ID
  - **A**: action (operation) performed (like «insert tuple t»)
  - **B**: block where A was performed
- For each action there is a compensating action that cancels the effect (like delete for insert) and can be constructed based upon A
- If T aborts, all T´s actions are compensated, and the compensation is logged
- Each block has the log record number of the last action that affected it