

## *Schedules involving abort*

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

This is an  
unrecoverable  
schedule

- ❖ Recoverable schedule: transactions commit only after all transactions whose changes they read commit.

# Recoverable schedules

- ❖ We must also consider the impact of transaction failures on concurrently running transactions.
  - That is, schedules with ABORT
- ❖ **Recoverable schedule:** For any transactions  $T_i$  and  $T_j$ : if  $T_j$  reads data written by  $T_i$ , then  $T_i$  commits before  $T_j$  commits.

A Non-recoverable schedule.

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

DBMS must ensure recoverable schedules.

## *Cascadeless schedules*

- ❖ Even if schedule is recoverable, several transactions may need to be rolled back to recover correctly.
- ❖ **Cascading Rollback:** a single transaction failure leading to a series of rollbacks

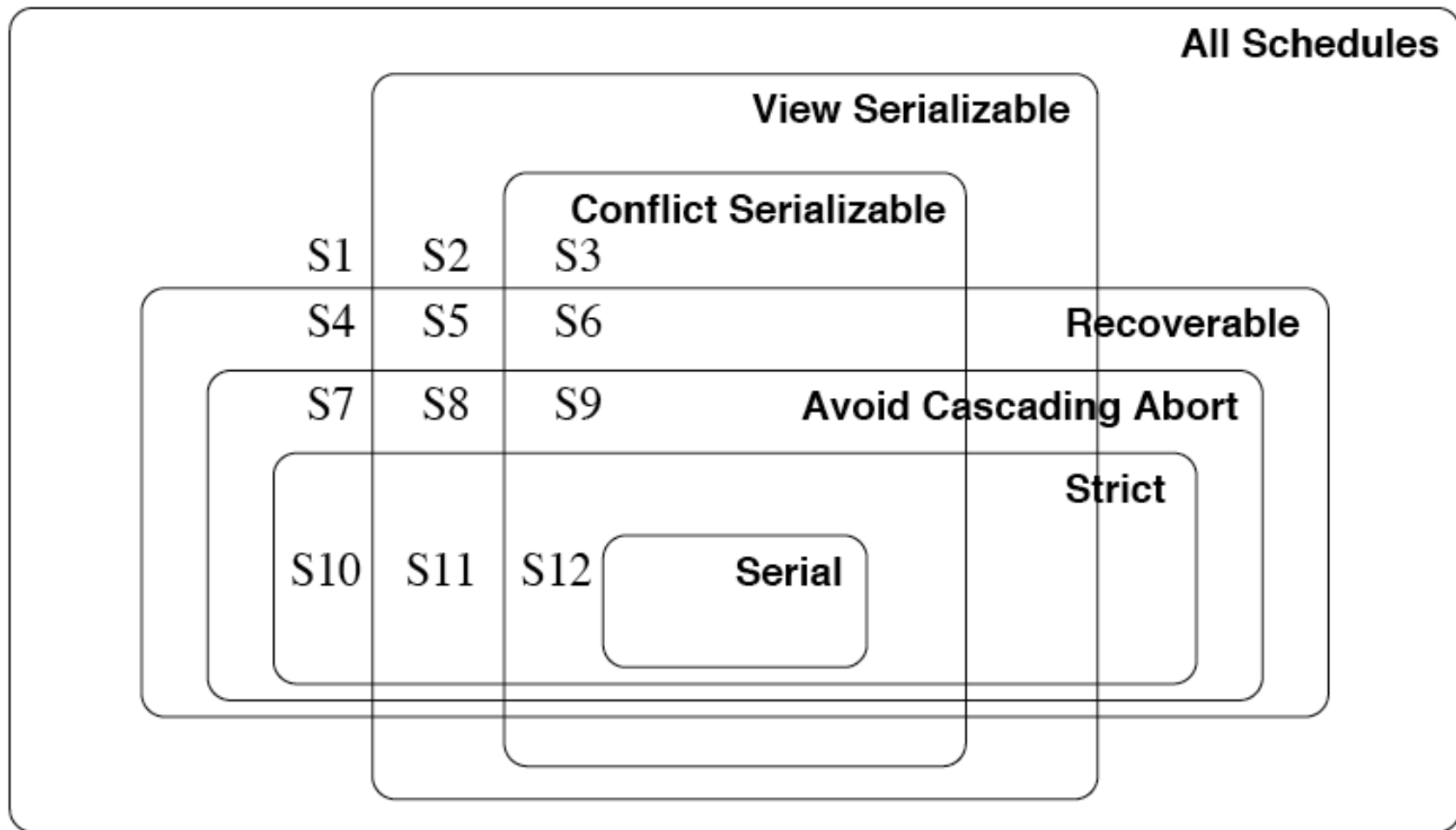
T1	T2	T3
R(A) R(B) W(A)	R(A) W(A)	R(A)
Abort		

- ❖ **Cascadeless schedule:** For any transactions  $T_i$  and  $T_j$ : if  $T_j$  reads data written by  $T_i$ , then  *$T_i$  commits before read operation of  $T_j$ .*

# *Strict schedules*

- ❖ A schedule is **strict** if:
  - A value written by a transaction T is not read or overwritten by other transactions until T either aborts or commits.
- ❖ Strict schedules are recoverable and cascadeless.

# *Properties of schedules*



# *Concurrency control schemes*

- ❖ The DBMS must provide a mechanism that will ensure all possible schedules are:
  - serializable
  - recoverable, and preferably cascadeless
- ❖ Concurrency control protocols ensure these properties.

# *Lock-Based Concurrency Control*

- ❖ Lock - associated with some object
  - shared or exclusive
- ❖ Locking protocol - set of rules to be followed by each transaction to ensure good properties.

# Lock Compatibility Matrix

Locks on a data item are granted based on a  
lock compatibility matrix:

		Mode of Data Item		
		None	Shared	Exclusive
Request mode {	Shared	Y	Y	N
	Exclusive	Y	N	N

When a transaction requests a lock, it must wait  
(block) until the lock is granted



# *Transaction performing locking*

T1
lock-X(A) <b>R(A)</b> <b>W(A)</b> unlock(A) lock-S(B) <b>R(B)</b> unlock(B)

# *Two-Phase Locking (2PL)*

## ❖ Two-Phase Locking Protocol

- Each Xact must obtain a *S (shared)* lock on object before reading, and an *X (exclusive)* lock on object before writing.
- A transaction can not request additional locks once it releases any locks.
  - This implies two phases:
    - *growing phase*
    - *shrinking phase*

# *Strict Two-Phase Locking (Strict 2PL)*

## ❖ Strict Two-phase Locking Protocol:

- Each Xact must obtain a *S (shared)* lock on object before reading, and an *X (exclusive)* lock on object before writing.
- A transaction can not request additional locks once it releases any locks.
- All X (exclusive) locks acquired by a transaction must be held until completion (commit/abort).

*Not admissible under 2PL*

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

# *Lock-based protocols*

- ❖ 2PL ensures **conflict serializability**
  - Transactions can be ordered by their end of growing phase (called **lock point**)
  - A 2PL schedule is equivalent to the serial schedule where transactions ordered by lock point order.
- ❖ Strict 2PL ensures **conflict serializable** and **cascadeless** schedules
  - Writers hold an X lock until they commit.

## *Schedule following strict 2PL*

T1	T2
S(A) <b>R(A)</b>	
	S(A) <b>R(A)</b> X(B) <b>R(B)</b> <b>W(B)</b> <b>Commit</b>
X(C) <b>R(C)</b> <b>W(C)</b> <b>Commit</b>	

# *Lock Management*

- ❖ Lock and unlock requests are handled by the lock manager
- ❖ Lock table entry (for an object):
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- ❖ Locking and unlocking have to be atomic operations
- ❖ Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

# Deadlock

T1	T2
X(A)	
	X(B)
X(B)	
	X(A)

**granted**

**granted**

**queued**

**queued**

- ❖ Deadlock: Cycle of transactions waiting for locks to be released by each other.



# *Deadlocks*

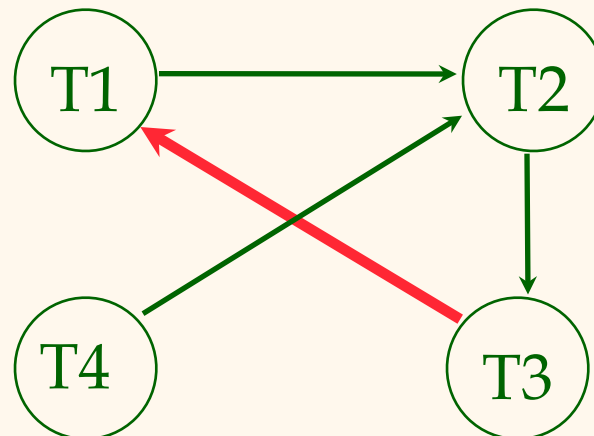
- ❖ Tend to be rare in practice.
- ❖ Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

# *Deadlock Detection*

- ❖ Create a **waits-for graph**:
  - Nodes are transactions
  - There is an edge from  $T_i$  to  $T_j$  if  $T_i$  is waiting for  $T_j$  to release a lock
  - add edge when queueing a lock request,
  - remove edge when granting lock request.
- ❖ Periodically check for cycles in the waits-for graph
- ❖ Resolve by aborting a transaction on cycle, releasing its locks.

## *Deadlock Detection (Continued)*

T1	T2	T3	T4
S(A) R(A)	X(B) W(B)		
S(B)		S(C) R(C)	
	X(C)		
		X(A)	X(B)



# *Deadlock Prevention*

- ❖ Assign priorities based on timestamps.  
Assume  $T_i$  wants a lock that  $T_j$  holds. Two policies are possible:
  - **Wait-Die:** If  $T_i$  has higher priority,  $T_i$  waits for  $T_j$ ; otherwise  $T_i$  aborts
  - **Wound-wait:** If  $T_i$  has higher priority,  $T_j$  aborts; otherwise  $T_i$  waits
- ❖ If a transaction re-starts, make sure it has its original timestamp (to avoid **starvation** of a transaction).

# *Performance of Locking*

- ❖ Lock-based schemes resolve conflicting schedules by **blocking** and **aborting**
  - in practice few deadlocks and relatively few aborts
  - most of penalty from blocking
- ❖ To increase throughput
  - lock smallest objects possible
  - reduce time locks are held
  - reduce hotspots

# *Anomalies with Interleaved Execution*

- ❖ Not all interleavings of operations are okay.
- ❖ **Anomaly**: two consistency-preserving committed transactions that lead to an inconsistent state.
- ❖ Types of anomalies:
  - Reading Uncommitted Data (WR Conflicts)  
“dirty reads”
  - Unrepeatable Reads (RW Conflicts)
  - Overwriting Uncommitted Data (WW Conflicts)

# *Reading Uncommitted Data*

“Dirty Read”

Inconsistent result of A is  
exposed to transaction T2

T1: Transfer	T2: Interest
R(A) W(A)	
	R(A) W(A) R(B) W(B) Commit
R(B) W(B) Commit	

# *Unrepeatable Reads*

T1 could see two values for A,  
although it has not changed A itself.

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



# *Overwriting Uncommitted Data*

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

# *Transaction support in SQL*

- ❖ Transaction automatically started for SELECT, UPDATE, CREATE
- ❖ Transaction ends with COMMIT or ROLLBACK (abort)
- ❖ SQL 99 supports SAVEPOINTS which are simple nested transactions

# *Specify isolation level*

- General rules of thumb w.r.t. isolation:
  - Fully serializable isolation is more expensive than “no isolation”
    - We can’t do as many things concurrently (or we have to undo them frequently)
- ❖ For performance, we generally want to specify the most relaxed **isolation level** that’s acceptable
  - Note that we’re “slightly” violating a correctness constraint to get performance!

# *Specifying isolation level in SQL*

SET TRANSACTION [READ WRITE | READ ONLY]  
ISOLATION LEVEL [LEVEL];

LEVEL =	SERIALIZABLE	↓ Less isolation
	REPEATABLE READ	
	READ COMMITTED	
	READ UNCOMMITTED	

The default isolation level is **SERIALIZABLE**

# *REPEATABLE READ*

- ❖ T reads only changes made by committed transactions
- ❖ No value read/written by T is changed by another transaction until T completes.

# *READ COMMITTED*

- ❖ T reads only changes made by committed transactions
- ❖ No value ~~read~~/written by T is changed by another transaction until T completes.
- ❖ Value read by T may be modified while T in progress.

X locks on written objects, held to end  
S locks on read objects, but released immediately.

# *READ UNCOMMITTED*

- ❖ Greatest exposure to other transactions
- ❖ Dirty reads possible
- ❖ Can't make changes: must be READ ONLY
- ❖ Does not obtain shared locks before reading
  - Thus no locks ever requested.

# *Summary of Isolation Levels*

Level	Dirty Read	Unrepeatable Read
READ UN-COMMITTED	Maybe	Maybe
READ COMMITTED	No	Maybe
REPEATABLE READ	No	No
SERIALIZABLE	No	No



# *Summary*

- ❖ Concurrency control and recovery are among the most important functions provided by a DBMS.
- ❖ Users need not worry about concurrency.
  - System guarantees nice properties: ACID
  - This is implemented using a locking protocol
- Users can trade isolation for performance using SQL commands