Database Management Systems

Lecture 2

Transactions. Concurrency Control

Scheduling Transactions

- ***schedule***
  - a list of operations (Read / Write / Commit / Abort) of *n* transactions with the property that the order of the operations in each individual transaction is preserved

Scheduling Transactions

T1	T2	Schedule
read(V)		read1(V)
read(sum)		read1(sum)
	read(V)	read2(V)
	V := V + 50	
	write(V)	write2(V)
	commit	commit2
read(V)		read1(V)
sum := sum + V		
write(sum)		write1(sum)
commit		commit1

Serial and Non-Serial Schedules

- ***serial schedule***
  - a schedule in which the actions of different transactions are not interleaved

T1	T2
	read(V)
	V := V + 50
	write(V)
	commit
read(V)	
read(sum)	
read(V)	
sum := sum + V	
write(sum)	
commit	

Serial and Non-Serial Schedules

- *non-serial schedule*
  - a schedule in which the actions of different transactions are interleaved

```
read1(V)
read1(sum)
read2(V)

write2(V)
commit2
read1(V)

write1(sum)
commit1
```

Serializability

- C – set of transactions
- $Sch(C)$ – the set of schedules for C
- <u>*serializable schedule*</u>
    - a schedule $S \in Sch(C)$ is serializable if the effect of S on a consistent database instance is identical to the effect of some serial schedule $S_0 \in Sch(C)$

Serializability

- serializability
  - correctness criterion for an interleaved execution schedule
    - consider the serial execution schedule (T_1, T_2, …, T_n), $T_i \in C$
      - assume the database instance is in a correct state prior to executing T_1
      - every transaction must preserve the consistency of the database

    => the database is in a correct state after T_n completes execution

    => if a serializable schedule is executed on a correct database instance, it produces a correct database instance (since it is equivalent to some serial schedule)
    - ensuring serializability prevents inconsistencies generated by concurrent transactions that interfere with one another

Serializability

- serializability
  - objective
    - finding interleaved schedules enabling transactions to execute concurrently without interfering with each other (such schedules result in a correct database state)

Serializability

- the order of *read* and *write* operations is important
- actions that cannot be swapped in a schedule:
    - actions belonging to the same transaction
    - actions in different transactions operating on the same object if at least one of them is a *write*

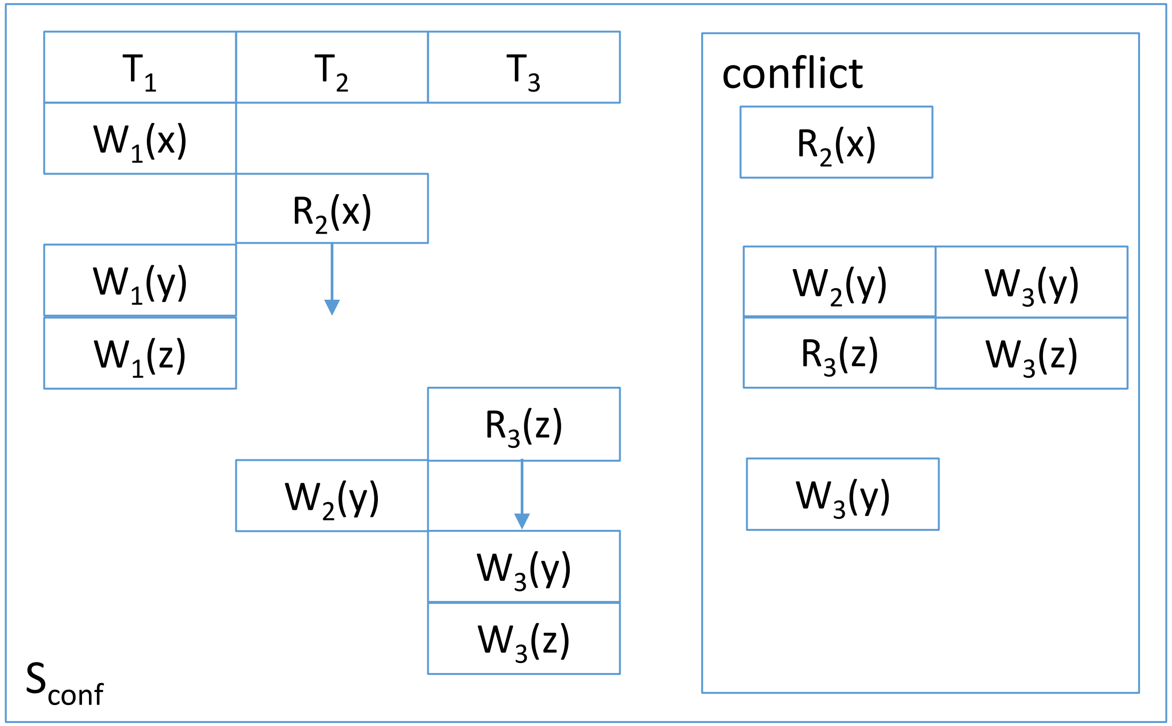
Conflict Serializability

- C – set of transactions
- $Sch(C)$ – the set of schedules for C
- $Op(C)$ – set of operations of the transactions in C
- consider schedule $S \in Sch(C)$
- the *conflict relation* of S is defined as:
  - $conflict(S)$ = {(op_1, op_2) | op_1, $op_2 \in Op(C)$, op_1 occurs before op_2 in S, op_1 and op_2 are in conflict}
- two schedules S_1 and $S_2 \in Sch(C)$ are <u>*conflict equivalent*</u>, written $S_1 \equiv_c S_2$, if $conflict(S_1) = conflict(S_2)$, i.e.:
  - S_1 and S_2 contain the same operations of the same transactions and
  - every pair of conflicting operations is ordered in the same manner in S_1 and S_2

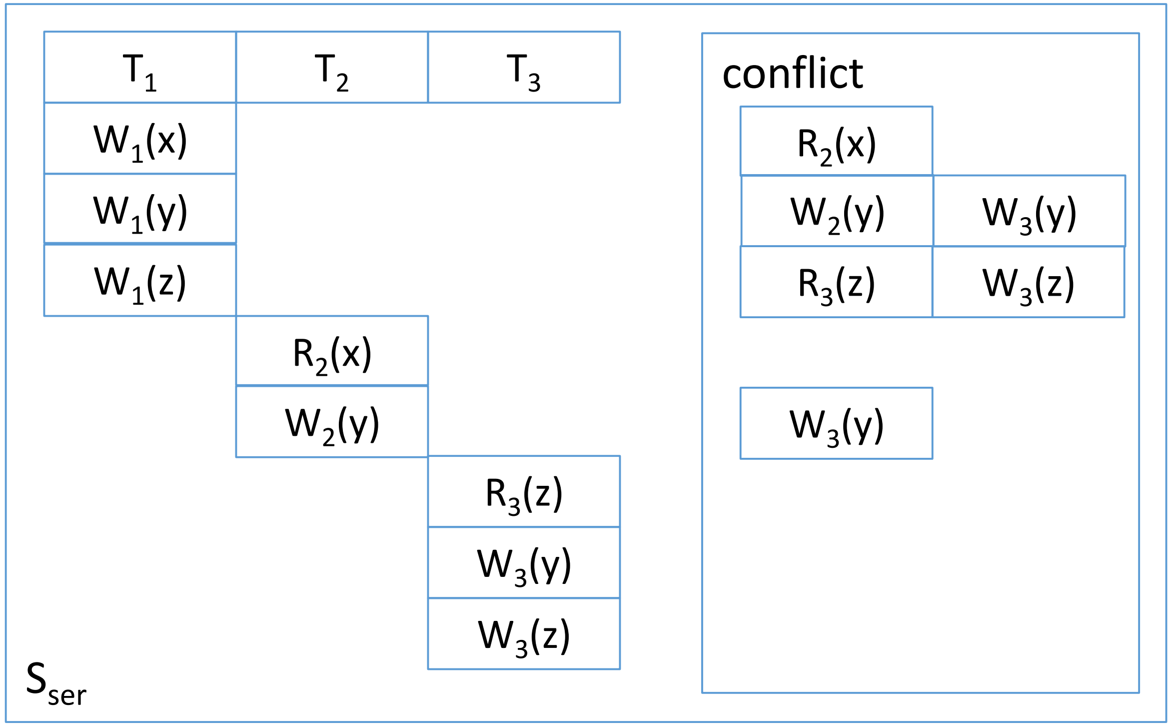
Conflict Serializability

- C – set of transactions
- $Sch(C)$ – the set of schedules for C
- let S be a schedule in $Sch(C)$
- schedule S is *conflict serializable* if there exists a serial schedule $S_0 \in Sch(C)$ such that $S \equiv_c S_0$, i.e., S is conflict equivalent to some serial schedule

Conflict Serializability



conflict serializable schedule



serial schedule

Conflict Serializability - Precedence Graph

- let *S* be a schedule in *Sch(C)*
- the *precedence graph* (*serializability graph*) of *S* contains:
  - one node for every committed transaction in *S*
  - an arc from T_i to T_j if an action in T_i precedes and conflicts with one of the actions in T_j

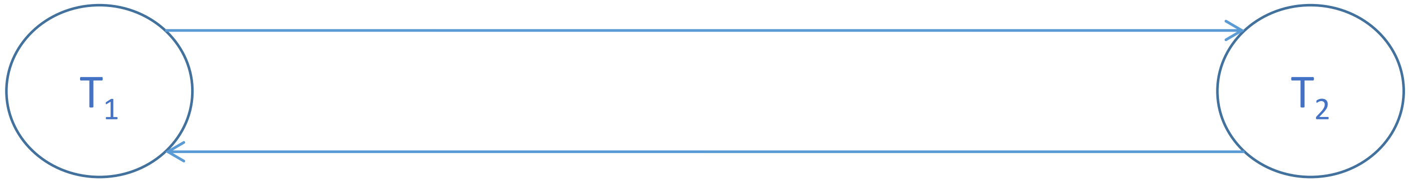
- Theorem:
  - a schedule $S \in Sch(C)$ is conflict serializable if and only if its precedence graph is acyclic

Conflict Serializability - Precedence Graph

- example - a schedule that is not conflict serializable:

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

- the precedence graph has a cycle:



Conflict Serializability - Precedence Graph

- algorithm to test the conflict serializability of a schedule $S \in Sch(C)$

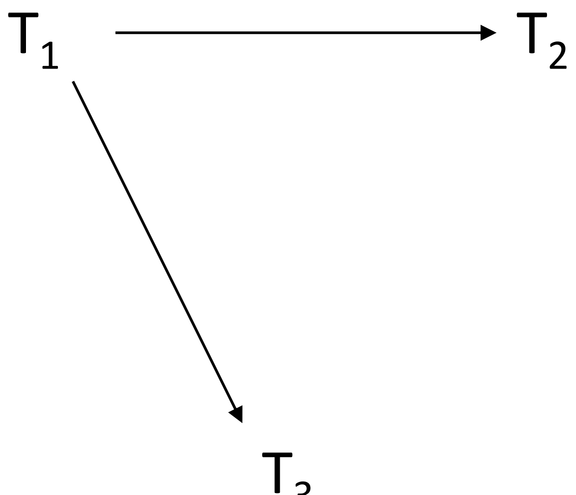
1. create a node labeled T_i in the precedence graph for every committed transaction T_i in the schedule
2. create an arc (T_i,T_j) in the precedence graph if T_j executes a Read(A) after a Write(A) executed by T_i
3. create an arc (T_i,T_j) in the precedence graph if T_j executes a Write(A) after a Read(A) executed by T_i
4. create an arc (T_i,T_j) in the precedence graph if T_j executes a Write(A) after a Write(A) executed by T_i
5. *S* is conflict serializable if and only if the resulting precedence graph has no cycles

Conflict Serializability - Precedence Graph

- examples
- let S_1 be a schedule over $\{T_1, T_2, T_3\}$:

T_1	T_2	T_3
$R_1(x)$		
$R_1(y)$		
	$R_2(y)$	
$W_1(x)$		
		$R_3(x)$
	$W_2(y)$	
		$W_3(x)$

- the precedence graph for S_1:

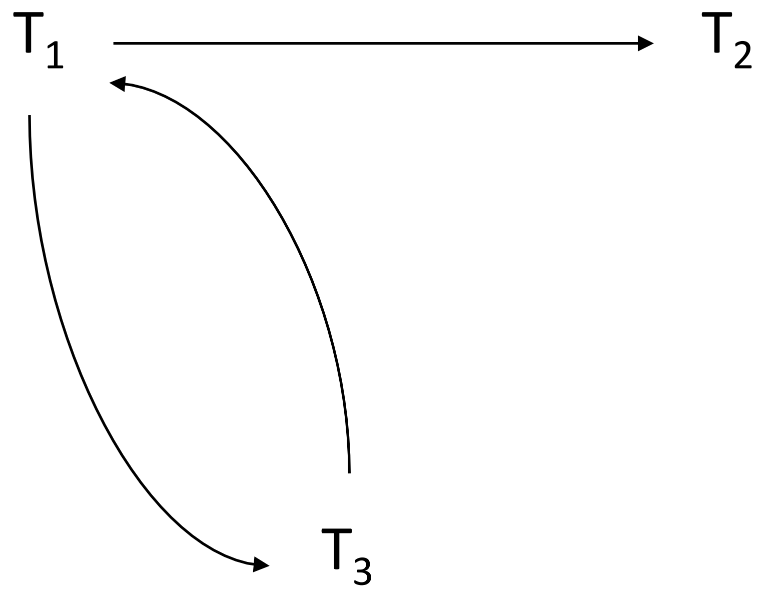


Conflict Serializability - Precedence Graph

- examples
- let S_2 be a schedule over $\{T_1, T_2, T_3\}$:

T_1	T_2	T_3
$R_1(x)$		
$R_1(y)$		
		$R_3(x)$
	$R_2(y)$	
$W_1(x)$		
	$W_2(y)$	
		$W_3(x)$

- the precedence graph for S_2:



Conflict Serializability

- every conflict serializable schedule is serializable (in the absence of inserts / deletes, when items can only be updated)
- there are serializable schedules that are not conflict serializable
- *S* is equivalent to the serial execution of transactions T_1, T_2, T_3 (in this order), but it is not conflict equivalent to this serial schedule (the write operations in T_1 and T_2 are ordered differently)

S

T_1	T_2	T_3
$R_1(A)$		
	$W_2(A)$	
	commit	
$W_1(A)$		
commit		
		$W_3(A)$
		commit

View Serializability

- conflict serializability is a sufficient condition for serializability, but it is not a necessary one
- *view serializability*
  - a more general, sufficient condition for serializability
  - based on *view-equivalence,* a less stringent form of equivalence

View Serializability

- C – set of transactions
- $Sch(C)$ – the set of schedules for C
- let T_i, $T_j \in C$, S_1, $S_2 \in Sch(C)$; S_1 and S_2 are <u>*view equivalent*</u>, written $S_1 \equiv_v S_2$, if the following conditions are met:
  - if T_i reads the initial value of V in S_1, then T_i also reads the initial value of V in S_2;
  - if T_i reads the value of V written by T_j in S_1, then T_i also reads the value of V written by T_j in S_2;
  - if T_i writes the final value of V in S_1, then T_i also writes the final value of V in S_2.
- i.e.:

each transaction performs the same computation in S_1 and S_2

and

S_1 and S_2 produce the same final database state.

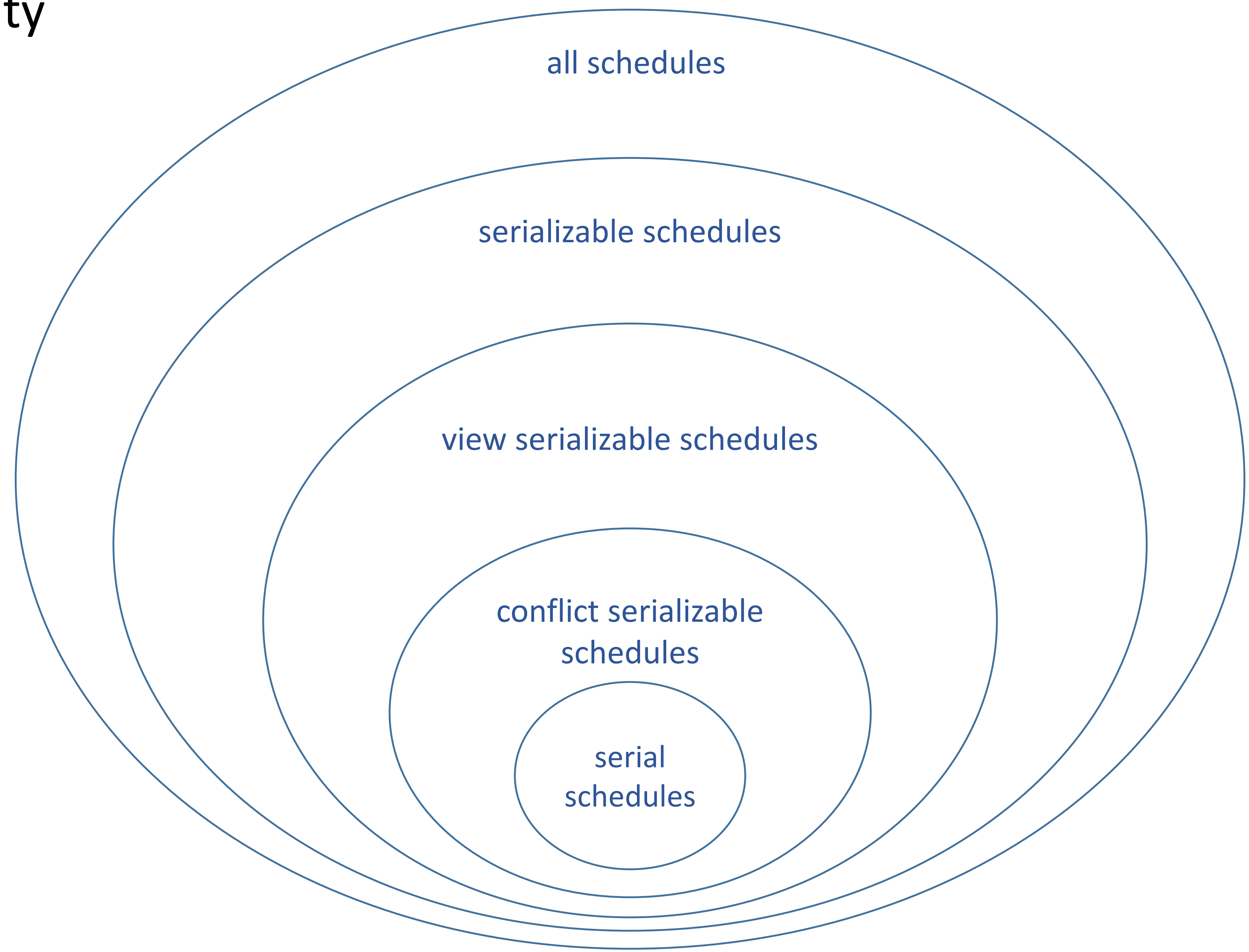
View Serializability

- C – set of transactions
- $Sch(C)$ – the set of schedules for C
- a schedule $S \in Sch(C)$ is ***view serializable*** if there exists a serial schedule $S_0 \in Sch(C)$ such that $S \equiv_v S_0$, i.e., S is view equivalent to some serial schedule

T_1	T_2	T_3
$R_1(A)$		
	$W_2(A)$	
	commit	
$W_1(A)$		
commit		
		$W_3(A)$
		commit

T_1	T_2	T_3
$R_1(A)$		
$W_1(A)$		
commit		
	$W_2(A)$	
	commit	
		$W_3(A)$
		commit

Serializability



Recoverable Schedules

- consider schedule *S* over $\{T_1, T_2\}$

T_1	T_2
Read(x)	
x := x + 10	
Write(x)	
	Read(x)
	x := x * 5
	Write(x)
	Read(y)
	y := y * 5
	Write(y)
	commit
abort	

- T_2 operates on a value of x that shouldn't have been in the database, since T_1 aborted

Recoverable Schedules

T_1	T_2
Read(x)	
x := x + 10	
Write(x)	
	Read(x)
	x := x * 5
	Write(x)
	Read(y)
	y := y * 5
	Write(y)
	commit
abort	

- cannot cascade the abort of T_1, since T_2 has already committed
- schedule *S* is *unrecoverable*

Recoverable Schedules

- recoverable schedule
  - a schedule in which a transaction T commits only after all transactions whose changes T read commit

Avoiding Cascading Aborts

T_1	T_2
Write(x)	
	Read(x)
	Write(y)
abort	
	=> abort

- a schedule in which a transaction T is reading only changes of committed transactions is said to <u>avoid cascading aborts</u>
- avoiding cascading aborts => recoverable schedules

Lock-Based Concurrency Control

- technique used to guarantee serializable, recoverable schedules
- *lock*
  - a tool used by the transaction manager to control concurrent access to data
  - prevents a transaction from accessing a data object while another transaction is accessing the object
- *transaction protocol*
  - a set of rules enforced by the transaction manager and obeyed by all transactions
  - example – simple protocol: before a transaction can read / write an object, it must acquire an appropriate lock on the object
  - locks in conjunction with transaction protocols allow interleaved executions

Lock-Based Concurrency Control

- *transaction protocol*
  - it's impractical for the DBMS to test the serializability of schedules, since the operating system could determine the interleaving of operations
  - instead, the DBMS uses protocols known to produce serializable schedules

Lock-Based Concurrency Control

- <u>*SLock*</u> (*shared* or *read lock*)
  - if a transaction holds an SLock on an object, it can read the object, but it cannot modify it
- <u>*XLock*</u> (*exclusive* or *write lock*)
  - if a transaction holds an XLock on an object, it can both read and write the object
- if a transaction holds an SLock on an object, other transactions can be granted SLocks on the object, but they cannot acquire XLocks on it
- if a transaction holds an XLock on an object, other transactions cannot be granted either SLocks or XLocks on the object

	Shared	Exclusive
Shared	Yes	No
Exclusive	No	No

Lock-Based Concurrency Control

- lock upgrade
  - an SLock granted to a transaction can be upgraded to an XLock
- transactions are issuing lock requests to the lock manager
- locks are held until being explicitly released by transactions
- lock acquire / lock release requests are automatically inserted into transactions by the DBMS (not the user’s responsibility)
- locking / unlocking
  - atomic operations

Lock-Based Concurrency Control

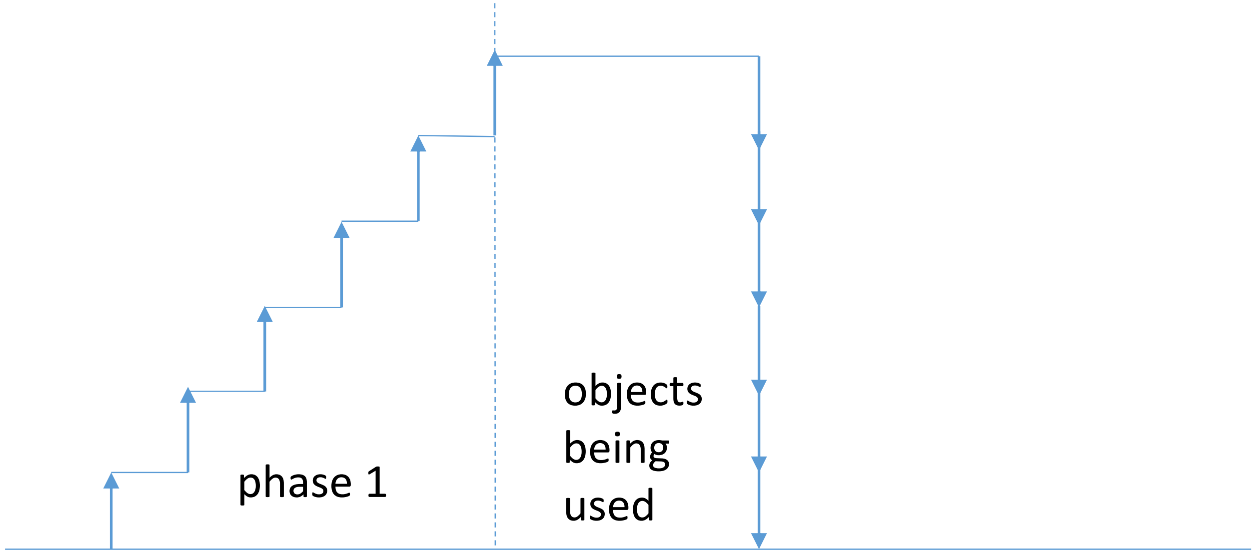
- *lock table*
  - structure used by the lock manager to keep track of granted locks / lock requests
  - entry in the lock table (corresponding to one data object):
    - number of transactions holding a lock on the data object
    - lock type (SLock / XLock)
    - pointer to a queue of lock requests

Lock-Based Concurrency Control

- *transactions table*
  - structure maintained by the DBMS
  - one entry / transaction
  - keeps a list of locks held by every transaction

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

- all the locks held by a transaction are released when it completes execution
- the Strict 2PL protocol allows only serializable schedules



Strict Two-Phase Locking

T_1	T_2
R(CM)	
W(CM)	
	R(CM)
	W(CM)
	R(CS)
	W(CS)
	commit
R(CS)	
W(CS)	
commit	

T_1	T_2
XLock(CM)	
R(CM)	
W(CM)	
XLock(CS)	
R(CS)	
W(CS)	
commit	
	XLock(CM)
	R(CM)
	W(CM)
	XLock(CS)
	R(CS)
	W(CS)
	commit

Strict Two-Phase Locking

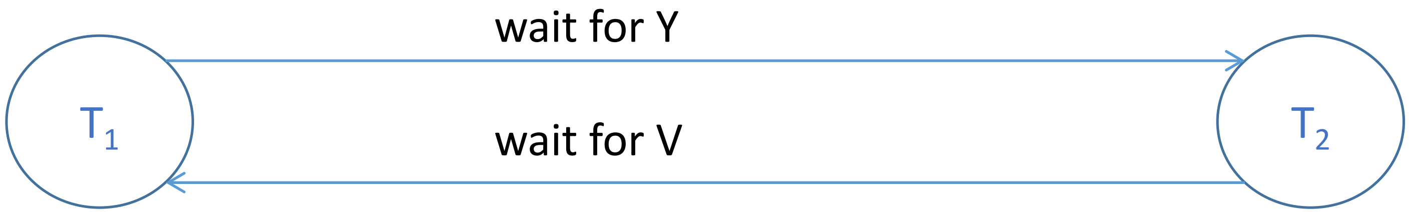
T_1	T_2
XLock(CM)	
R(CM)	
W(CM)	
	XLock(CT)
	R(CT)
	W(CT)
	commit
XLock(CS)	
R(CS)	
W(CS)	
commit	

Deadlocks

- lock-based concurrency control techniques can lead to deadlocks
- *deadlock*
  - cycle of transactions waiting for one another to release a locked resource
  - normal execution can no longer continue without an external intervention, i.e., deadlocked transactions cannot proceed until the deadlock is resolved
- deadlock management
  - deadlock prevention
  - deadlock detection
    - allow deadlocks to occur and resolve them when they arise

Deadlocks

- example



T1

BEGIN TRAN
XLock(V)
Read(V)
V := V + 100
Write(V)
XLock(Y)
Wait
Wait
…

T2

BEGIN TRAN
XLock(Y)
Read(Y)
Y = Y * 5
Write(Y)
XLock(V)
Wait
Wait
…

Deadlocks - Prevention

- assign transactions timestamp-based priorities
  - i.e., the older a transaction is, the higher its priority
- 2 policies: Wait-Die and Wound-Wait
- assume T_i wants to access an object locked by T_j
  - Wait-Die
    - if T_i's priority is higher, T_i can wait; otherwise, T_i is aborted
  - Wound-Wait
    - if T_i's priority is higher, T_j is aborted; otherwise, T_i can wait
- if an aborted transaction is restarted, it's assigned its original timestamp

Deadlocks - Prevention

- example – Wait-Die / Wound-Wait

		Wait-die		Wound-wait	
T1	T2	T1	T2	T1	T2
R1(x)		R1(x)		R1(x)	
	R2(y)		R2(y)		R2(y)
W1(y)		W1(y) (wait)		W1(y)	(Abort T2)
	W2(x)		W2(x) (A)		

References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, http://codex.cs.yale.edu/avi/db-book/
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html