Database Management Systems

Lecture 3

Transactions. Concurrency Control

- locking *protocols*
  - Strict Two-Phase Locking
  - Two-Phase Locking

- *deadlocks*
  - prevention (Wait-die, Wound-wait)
  - detection (waits-for graph, timeout mechanism)

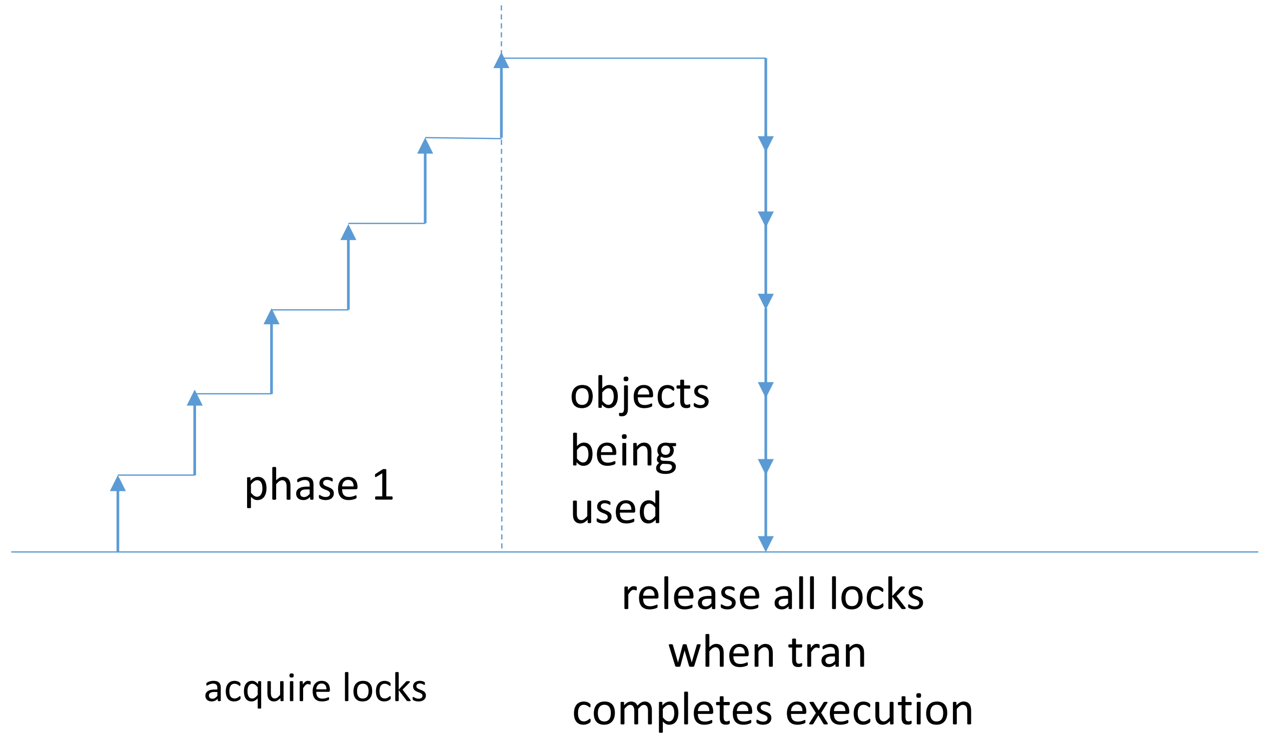
- the *phantom* problem

- *isolation levels*
  - read uncommitted
  - read committed
  - repeatable read
  - serializable

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

* before a transaction can read / write an object, it must acquire a S / X lock on the object

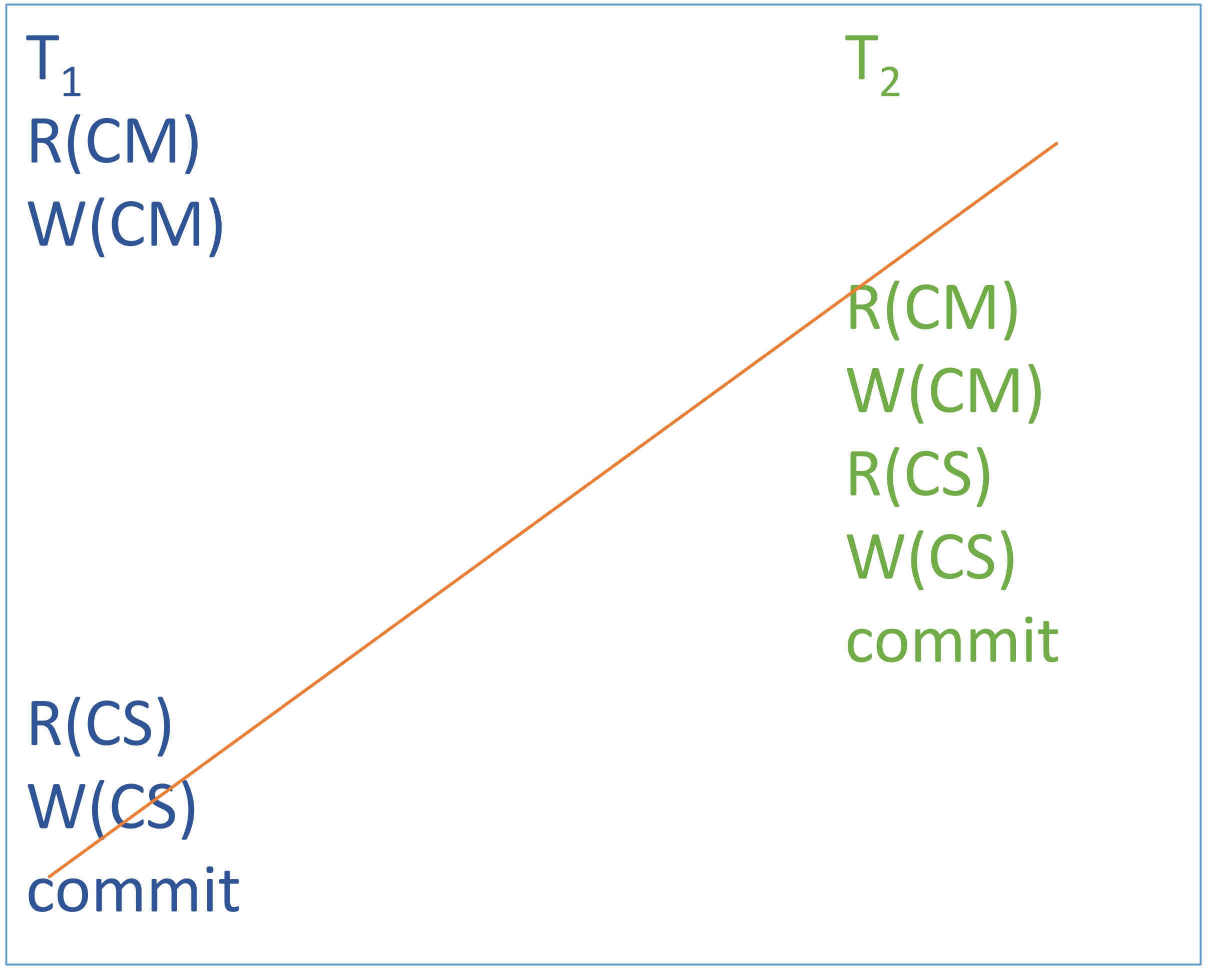
* all the locks held by a transaction are released when it completes execution



- the Strict 2PL protocol allows only serializable schedules (only schedules with acyclic precedence graphs are allowed by this protocol)

Strict Two-Phase Locking

- the interleaving below is not allowed by Strict 2PL:



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

->

Strict Two-Phase Locking

T_1	T_2
XLock(CM)	
R(CM)	
W(CM)	
	XLock(CM)
…	…

- T1 acquires an X lock on object CM, reads and writes CM
- T1 is still in progress when T2 requests a lock on the same object, CM
- T2 cannot acquire an exclusive lock on CM, since T1 already holds a conflicting lock on this object
- T1 will release its lock on CM only when it completes execution (with *commit* or *abort*)
- since it cannot grant T2 the requested lock on CM, the DBMS suspends T2

- in this example, we denote by *XLock(O)* the action of the current transaction requesting an X lock on object O

Strict Two-Phase Locking

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

- T1 continues execution
- when T1 commits, it releases both locks (X lock on CM, X lock on CS)
- T2 can now be granted an X lock on CM
- T2 can now proceed

Strict Two-Phase Locking

- the interleaving below is allowed by Strict 2PL:

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

- in this example, since T1 and T2 are operating on separate data objects (CM, CT, CS), they can concurrently obtain all requested locks (same would be true if T1 and T2 had, say, read the same data object A)

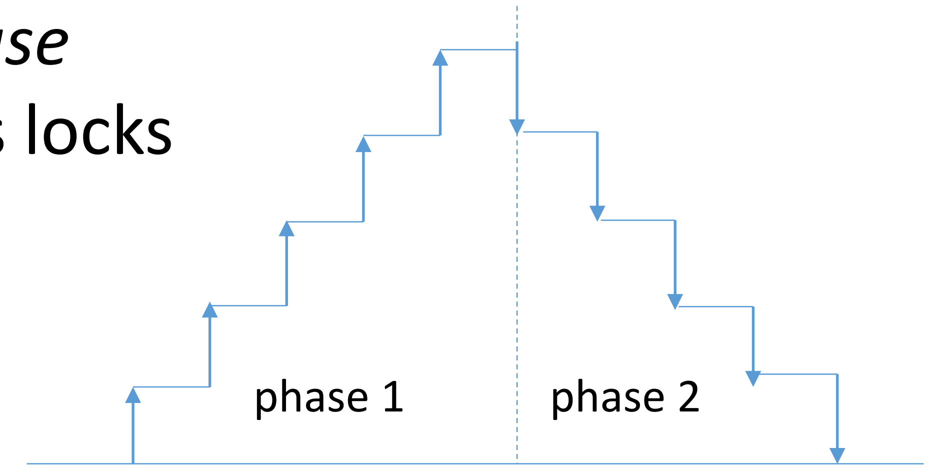
Two-Phase Locking (2PL)

- variant of Strict Two-Phase Locking

  * before a transaction can read / write an object, it must acquire a S / X lock on the object

  * once a transaction releases a lock, it cannot request other locks

- phase 1 - *growing phase*
  - transaction acquires locks
- phase 2 - *shrinking phase*
  - transaction releases locks



Two-Phase Locking

- *C* – set of transactions
- *Sch(C)* – set of schedules for *C*
- if all transactions in *C* obey 2PL, then any schedule $S \in Sch(C)$ that completes normally is serializable

Two-Phase Locking

- the following execution is allowed by the protocol:

- T1 can release its X lock on A prior to completion
- so T2 can acquire an X lock on A while T1 is still in progress
- however, T1 cannot acquire any other locks once it released a lock (in this case, its X lock on A)

- notation: *Release(O)* - the current transaction releases its lock on object O

T1	T2
XLock(A)	
XLock(B)	
R(A)	
A := A + 100	
W(A)	
Release(A)	
	XLock(A)
	XLock(C)
	R(A)
	A := A + 200
	W(A)
	R(C)
	C := C + 200
	W(C)
	Release(A)
	Release(C)
…	Commit
R(B)	
B := B + 200	
W(B)	
Release(B)	
Commit	

Two-Phase Locking

T1	T2	Values
XLock(A)		
XLock(B)		
R(A)		100
A := A + 100		
W(A)		200
Release(A)		
	XLock(A)	
	XLock(C)	
	R(A)	200
	A := A + 200	
	W(A)	400
	R(C)	
	C := C + 200	
	W(C)	
	Release(A)	
	Release(C)	
	Commit	
…		

t

- suppose T1 is forced to terminate at time t

=> T1's updates are undone (value of A is restored to 100)

=> T2's update to A is lost (incorrect, as atomicity is compromised, T2 also changed the value of C)

- problem – T1 released its exclusive lock on A prior to completion (under Strict 2PL, T1 can release its lock on A, as well as any other locks, only when it commits / aborts)

Strict Schedules

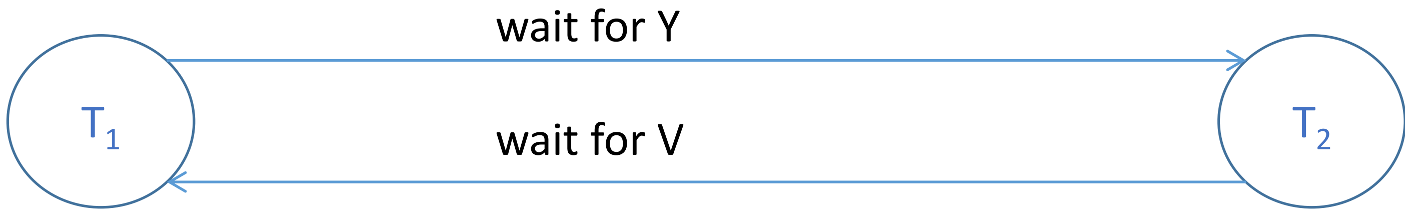
- if transaction T_i has written object A, then transaction T_j can read and / or write A only after T_i's completion (commit / abort)
- strict schedules:
    - avoid cascading aborts
    - are recoverable schedules
    - if a transaction is aborted, its operations can be undone

- Strict 2PL only allows strict schedules

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


$$\begin{tikzcd} T_1 \arrow[r, bend left, "\text{wait for Y}"] & T_2 \arrow[l, bend left, "\text{wait for V}"] \end{tikzcd}$$

```
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
…
```

- T1 cannot obtain an X lock on Y, since T2 holds a conflicting lock on Y
- similarly, T2 cannot obtain an X lock on V, since T1 holds a conflicting lock on V

Deadlocks - Prevention

- assign transactions timestamp-based priorities (each transaction has a timestamp - the moment it begins execution)
- the lower the timestamp, the older the transaction
- the older a transaction is, the higher its priority, with the oldest transaction having the highest priority

- 2 deadlock prevention policies: Wait-die and Wound-wait

Deadlocks - Prevention

- assume T_1 wants to access an object locked by T_2 (with a conflicting lock)
  - Wait-die
    - if T_1's priority is higher, T_1 can wait; otherwise, T_1 is aborted
- in the following execution, 2 transactions are reading and / or writing 2 objects, x and y; T1's priority is higher:

Wait-die

```
T1          T2                                      T1          T2
--------------------                                --------------------
R1(x)                                               R1(x)
            R2(y)                                               R2(y)
W1(y)                                               W1(y) (wait)
             W2(x)                                              W2(x) (A)
```

Deadlocks - Prevention

Wait-die

```
T1          T2                                         T1          T2
--------------------                                   --------------------
R1(x)                                                   R1(x)
            R2(y)                                                  R2(y)
W1(y)                                                   W1(y) (wait)
            W2(x)                                                  W2(x) (A)
```

- T1 requests an X lock on object y, which is already locked with a conflicting lock by T2
- since T1 has a higher priority, it is allowed to wait
- T2 asks for an X lock on object x, already locked with a conflicting lock by T1
- since T2 has a lower priority, it is aborted
- T1 now obtains the requested lock on object y and proceeds with the write operation

Deadlocks - Prevention

- assume T_1 wants to access an object locked by T_2 (with a conflicting lock)
  - Wound-wait
    - if T_1's priority is higher, T_2 is aborted; otherwise, T_1 can wait

Wound-wait

```
T1          T2                                   T1          T2
--------------------                             --------------------
R1(x)                                            R1(x)
            R2(y)                                            R2(y)
W1(y)                                            W1(y)       (Abort T2)
            W2(x)
```

- T1 requests an X lock on object y, which is already locked with a conflicting lock by T2
- since T1 has a higher priority, T2 is aborted
- T1 obtains the requested lock on object y and continues execution

Deadlocks - Prevention

- under these policies (Wait-die / Wound-wait), deadlock cycles cannot develop
- if an aborted transaction is restarted, it’s assigned its original timestamp

Deadlocks - Detection

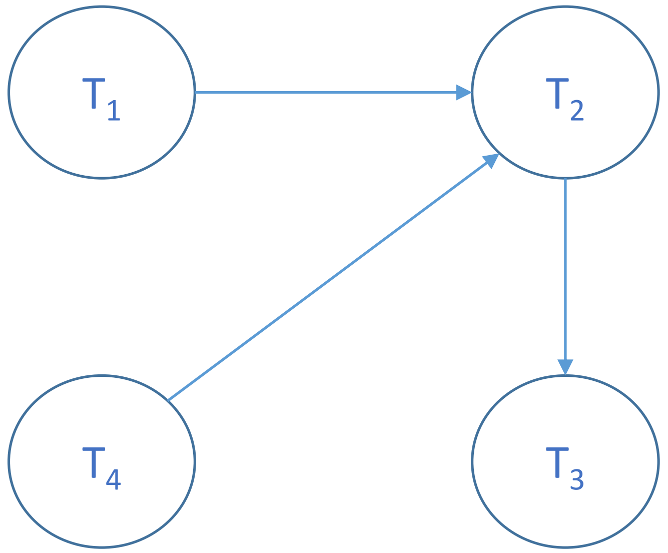
a. *waits-for graph*

- structure maintained by the lock manager to detect deadlock cycles
  - a node / active transaction
  - arc from T_i to T_j if T_i is waiting for T_j to release a lock
- cycle in the graph => deadlock
- DBMS periodically checks whether there are cycles in the waits-for graph

Deadlocks - Detection

a. ***waits-for graph***

T_1	T_2	T_3	T_4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)

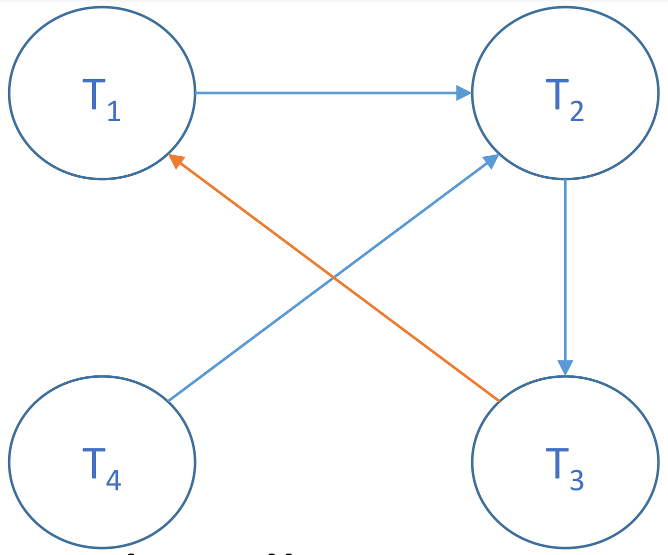


Deadlocks - Detection

a. *waits-for graph*

- if operation X(A) is also part of T3, there will be an arc from T3 to T1 in the graph (since T1 holds a conflicting lock on A, and T3 is waiting for T1 to release this lock)
- i.e., the graph contains a cycle (T1 is also waiting for T2 to release a lock, which in turn is waiting for T3 to release a lock)
  => deadlock

T_1	T_2	T_3	T_4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	

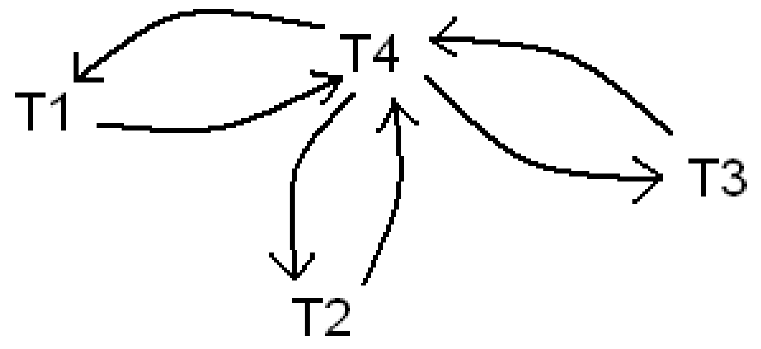


- aborting a transaction that appears on a cycle allows several other transactions to proceed

Deadlocks - Detection

a. *waits-for graph*

T_1	T_2	T_3	T_4
S(A)	S(A)	S(A)	
R(A)	R(A)	R(A)	
			S(B)
			R(B)
X(B)	X(B)	X(B)	
			X(A)
…	…	…	…



Deadlocks - Detection

b. *timeout mechanism*

- very simple, practical method of detecting deadlocks
- if a transaction T has been waiting too long for a lock on an object, a deadlock is assumed to exist and T is terminated

Deadlocks – Choosing the Deadlock Victim

- possible criteria to consider when choosing the deadlock victim
  - the number of objects modified by the transaction
  - the number of objects that are to be modified by the transaction
  - the number of locks held
- the policy should be “fair”, i.e., if a transaction is repeatedly chosen as a victim, it should be eventually allowed to proceed

The Phantom Problem

example 1. Researchers[RID, …, ImpactFactor, Age]

- Page1: <R1, 5, 30>, <R2, 5, 20>
- Page2: <R4, 5, 100>
- Page3: <R8, 6, 18>, <R9, 6, 19>

- concurrent transactions T1 and T2
  - transaction T1
    - retrieve the age of the oldest researcher for each of the impact factor values 5 and 6
  - transaction T2
    - add new researcher with impact factor 5
    - remove researcher R9

- T1 and T2 obey Strict 2PL

The Phantom Problem

- Page1: <R1, 5, 30>, <R2, 5, 20>
- Page2: <R4, 5, 100>
- Page3: <R8, 6, 18>, <R9, 6, 19>

T1	T2
SLock(Page1)	
SLock(Page2)	
compute max age for IF 5 => 100	
	XLock(Page4)
	XLock(Page3)
	add record <R5, 5, 102> to Page4
	delete researcher R9
	commit – all locks are released
SLock(Page3)	
compute max age for IF = 6 => 18	
…	

- T1 identifies and locks pages holding researchers with IF 5 (Page1, Page2)
- T1 computes max age for IF 5 (100)
- T2 can acquire X locks on two different pages: Page4 (to which it adds a new researcher with IF 5 and age 102) and Page3 (from which it deletes researcher R9, with IF 6 and age 19)
- T2 then commits, releasing all its locks
- T1 now obtains an S lock on Page3 (containing all researchers with IF 6), and computes max age for IF 6 (18)

The Phantom Problem

- Page1: <R1, 5, 30>, <R2, 5, 20>
- Page2: <R4, 5, 100>
- Page3: <R8, 6, 18>, <R9, 6, 19>

- outcome of interleaved schedule on the right:
  - IF 5, Max Age 100
  - IF 6, Max Age 18
- outcome of serial schedule (T1T2):
  - IF 5, Max Age 100
  - IF 6, Max Age 19
- outcome of serial schedule (T2T1):
  - IF 5, Max Age 102
  - IF 6, Max Age 18

T1	T2
SLock(Page1)	
SLock(Page2)	
compute max age for IF 5 => 100	
	XLock(Page4)
	XLock(Page3)
	add record <R5, 5, 102> to Page4
	delete researcher R9
	commit – all locks are released
SLock(Page3)	
compute max age for IF = 6 => 18	
…	

->

The Phantom Problem

=> the interleaved schedule is not serializable, as its outcome is not identical to the outcome of any serial schedule

- however, the schedule is conflict serializable (the precedence graph is acyclic)

=> in the presence of insert operations, i.e., if new objects can be added to the database, conflict serializability does not guarantee serializability

The Phantom Problem
example 2.

- T1 executes the same query twice
- between the 2 read operations, another transaction T2 inserts a row that meets the condition in T1's query; T2 commits

T1	T2	result set for T1's query
`SELECT * FROM Students WHERE GPA >= 8`		row corresponding to student with sid 12 is not in the result set
	`INSERT INTO Students VALUES (12, 'Mara', 'Dobse', 10)` COMMIT	
`SELECT * FROM Students WHERE GPA >= 8` …		row corresponding to student with sid 12 now appears in the result set

Transaction Support in SQL - Isolation Levels

- SQL provides support for users to specify various aspects related to transactions, e.g., isolation levels

- *isolation level*
  - a transaction’s characteristic
  - determines the degree to which a transaction is isolated from the changes made by other concurrently running transactions

- greater concurrency -> concurrency anomalies

Transaction Support in SQL - Isolation Levels

- 4 isolation levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

- isolation levels can be set with the following command:
  - SET TRANSACTION ISOLATION LEVEL *isolevel*
  - *e.g.,* SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

Isolation Levels

- READ UNCOMMITTED
  - a transaction must acquire an exclusive lock prior to writing an object
  - no locks are requested when reading objects
  - exclusive locks are released at the end of the transaction
  - lowest degree of isolation

T1	T2	Values
R(A)		10
W(A)		20
	R(A)	20
	…	
	Commit	
…		
Abort		

- dirty reads can occur under this isolation level
- T1 acquires an X lock on A, reads A (value 10), writes A (value 20)
- T1’s X lock on A will only be released when T1 commits / aborts
- T1 is still in progress when T2 attempts to read A; since no S lock is required when reading data under READ UNCOMMITTED, T2 is able to read value 20 for A

Isolation Levels

- READ UNCOMMITTED

unrepeatable reads ✓

T1	T2
R(A)	
	R(A)
	W(A)
	C
R(A)	
W(A)	
C	

phantoms ✓

T1	T2
R(students with id between 100 and 110)	
	insert student with id 101
	C
R(students with id between 100 and 110)	
C	

- it is easy to see that this isolation level also exposes transactions to unrepeatable reads and phantoms

Isolation Levels

- READ COMMITTED
    - a transaction must acquire an exclusive lock prior to writing an object
    - a transaction must acquire a shared lock prior to reading an object (i.e., the last transaction that modified the object is finished)
    - exclusive locks are released at the end of the transaction
    - shared locks are immediately released (as soon as the read operation is completed)

    - dirty reads can no longer occur under this isolation level, but unrepeatable reads and phantoms are still possible

Isolation Levels

- READ COMMITTED
  - dirty reads ✗ unrepeatable reads ✓ phantoms ✓

T1	T2
R(A)	
W(A)	
	R(A)
	…
	C
…	
A	

T1	T2
R(A)	
	R(A)
	W(A)
	C
R(A)	
W(A)	
C	

T1	T2
R(students with id between 100 and 110)	
	insert student with id 101
	C
R(students with id between 100 and 110)	
C	

Isolation Levels

- REPEATABLE READ
  - a transaction must acquire an exclusive lock prior to writing an object
  - a transaction must acquire a shared lock prior to reading an object
  - exclusive locks are released at the end of the transaction
  - shared locks are released at the end of the transaction

  - dirty reads and unrepeatable reads cannot occur under REPEATABLE READ, but phantoms are still possible

Isolation Levels

- REPEATABLE READ
  - dirty reads ✘ unrepeatable reads ✘ phantoms ✓

T1	T2
R(A)	
W(A)	
	R(A)
	…
	C
…	
A	

T1	T2
R(A)	
	R(A)
	W(A)
	C
R(A)	
W(A)	
C	

T1	T2
R(students with id between 100 and 110)	
	insert student with id 101
	C
R(students with id between 100 and 110)	
C	

Isolation Levels

- SERIALIZABLE
    - a transaction must acquire locks on objects before reading / writing them
    - a transaction can also acquire locks on sets of objects that must remain unmodified
        - if a transaction T reads a set of objects based on a search predicate, this set cannot be changed while T is in progress (if query *Return all students with GPA >= 8* is executed twice within a transaction, it must return the same answer set)
    - locks are held until the end of the transaction
    - highest degree of isolation

    - dirty reads, unrepeatable reads, phantoms can’t occur under this isolation level

Isolation Levels

- SERIALIZABLE
  - dirty reads ✘ unrepeatable reads ✘ phantoms ✘

T1	T2
R(A)	
W(A)	
	R(A)
	…
	C
…	
A	

T1	T2
R(A)	
	R(A)
	W(A)
	C
R(A)	
W(A)	
C	

T1	T2
R(students with id between 100 and 110)	
	insert student with id 101
	C
R(students with id between 100 and 110)	
C	

* Jim Gray *

primary research interests – databases, transaction processing systems

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
- [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