Database Applications (15-415)

DBMS Internals- Part XI
Lecture 19, April 2, 2014

Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

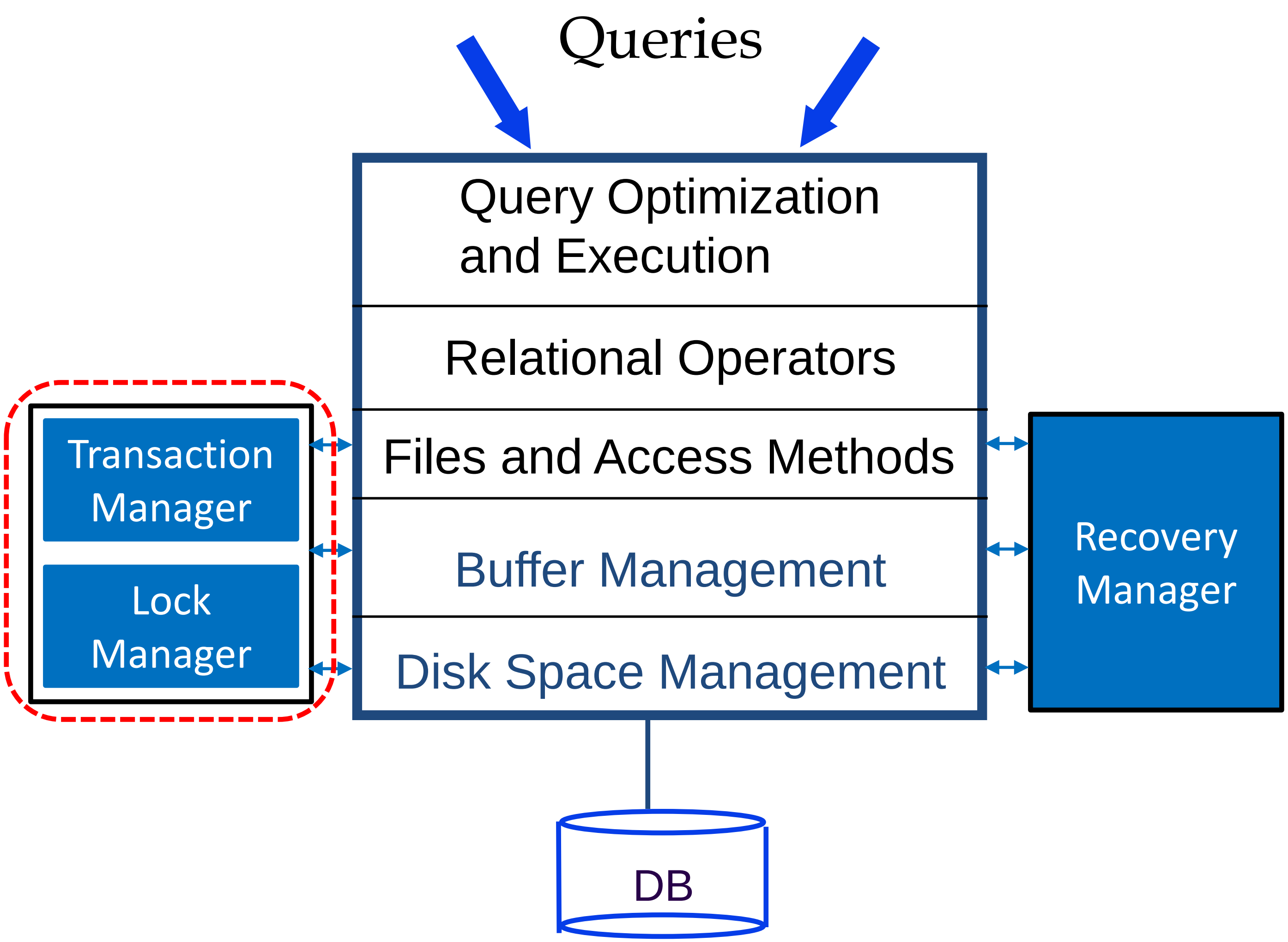
Today…

- Last Session:
  - DBMS Internals- Part IX
    - Query Optimization (*Cont'd*)
    - A "Very" Brief Introduction to Transaction Management

- Today's Session:
  - Transaction Management

- Announcements:
  - Quiz 2 is Tomorrow, April 3, at 5:00PM in Room 2147 (*all materials covered after the midterm are included, except transaction management*)
  - PS4 is now posted. It is due on Saturday, April 12th
  - Project 3 is due on Saturday, April 5th by midnight
  - On Monday, April 7th, every student will live demo his P3 in 5 minutes (during the class time)

DBMS Layers


Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction
Manager
Lock
Manager
Recovery
Manager
DB

Outline

- A Brief Primer on Transaction Management ✓
- Anomalies Due to Concurrency
- 2PL and Strict 2PL Locking Protocols
- Schedules with Aborted Transactions

Concurrent Execution of Programs

- A database is typically *shared* by a large number of users
- DBMSs *schedule* users' programs *concurrently*
  - While one user program is waiting for an I/O access to be satisfied, the CPU can process another program
    - Better system throughput
  - Interleaved execution of a short program with a long program allows the short program to complete quickly
    - Better response time
    - Better for fairness reasons

Transactions

- *Any <u>one</u> execution* of a user program in a DBMS is denoted as a transaction
  - Executing the same program several times will generate several transactions

- A transaction is the basic unit of change as seen by a DBMS
  - E.g., Transfer $100 from account A to account B

- A transaction may carry out many operations on data, but DBMSs are only concerned about *reads* and *writes*

- Thus, in essence a transaction becomes *a sequence of reads and writes*

Transactions (*Cont'd*)

- In addition to reading and writing, a transaction must specify as its final action:
  - Either *Commit* (i.e., complete successfully)
  - Or *Abort* (i.e., terminate and *undo* actions)

- We make two assumptions:
  - Transactions interact only via database reads and writes (i.e., no *message passing*)
  - A database is a fixed collection of *independent* objects (A, B, C, etc.)

Schedules

- A schedule is a list of actions (i.e., read, write, abort, and/or commit) from a *set* of transactions

- The *order* in which two actions of a transaction ***T*** appear in a schedule must be the same as they appear in ***T*** itself

- Assume **T1** = [R(A), W(A)] and **T2** = [R(B), W(B), R(C), W(C)]

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

✓

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

✓

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

✗

Serial Schedules

- A complete schedule must contain all the actions of every transaction that appears on it

- If the actions of different transactions are ***not interleaved***, the schedule is called a serial schedule

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

A Serial Schedule

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

A Non-Serial Schedule

Serializable Schedules

- Two schedules are said to be *equivalent* if for any database state, the effect of executing the 1st schedule is identical to the effect of executing the 2nd schedule

- A serializable schedule is a schedule that is equivalent to a serial schedule

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

A *Serializable* Schedule

Equivalent →

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

A *Serial* Schedule

← Equivalent

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

Another *Serializable* Schedule

Examples

- Assume transactions T1 and T2 as follows:

```
T1:     BEGIN   A=A-100,   B=B +100   END
T2:     BEGIN   A=1.06*A,   B=1.06*B   END
```

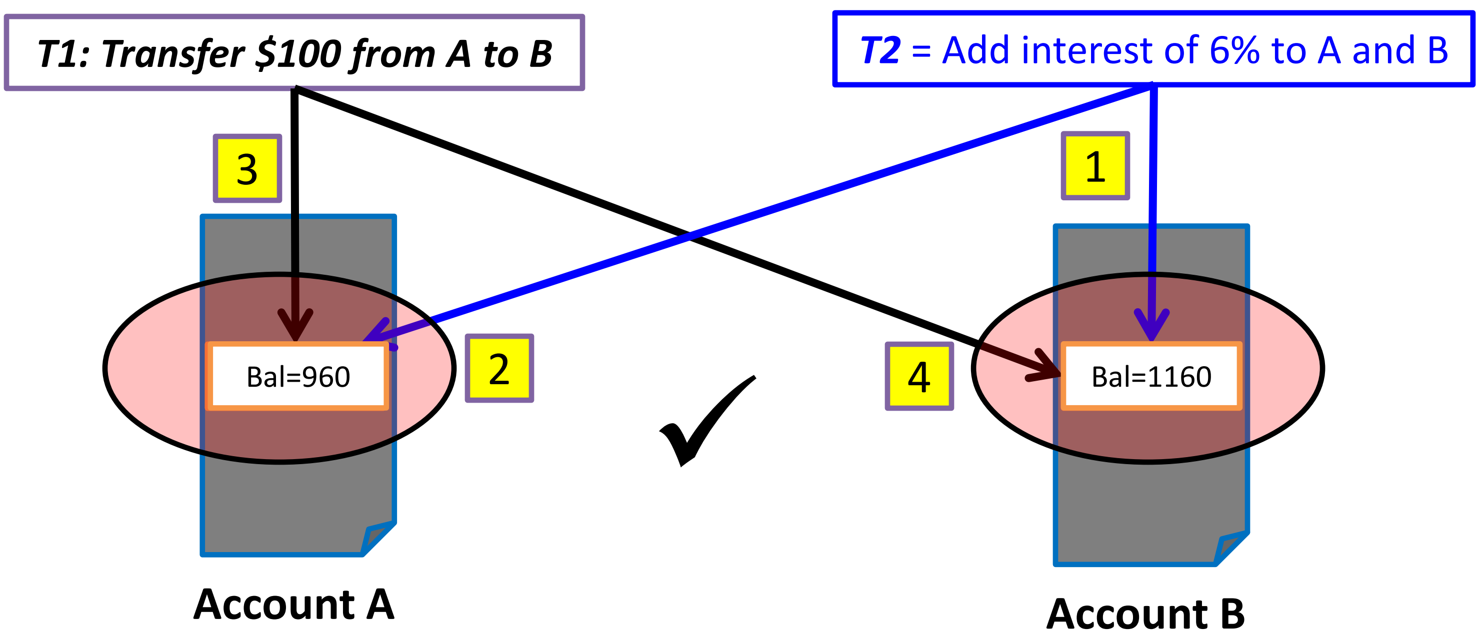
- T1 can be thought of as transferring $100 from A’s account to B’s account

- T2 can be thought of as crediting accounts A and B with a 6% interest payment

Examples: A *Serial* Schedule

- Assume transactions T1 and T2 as follows:

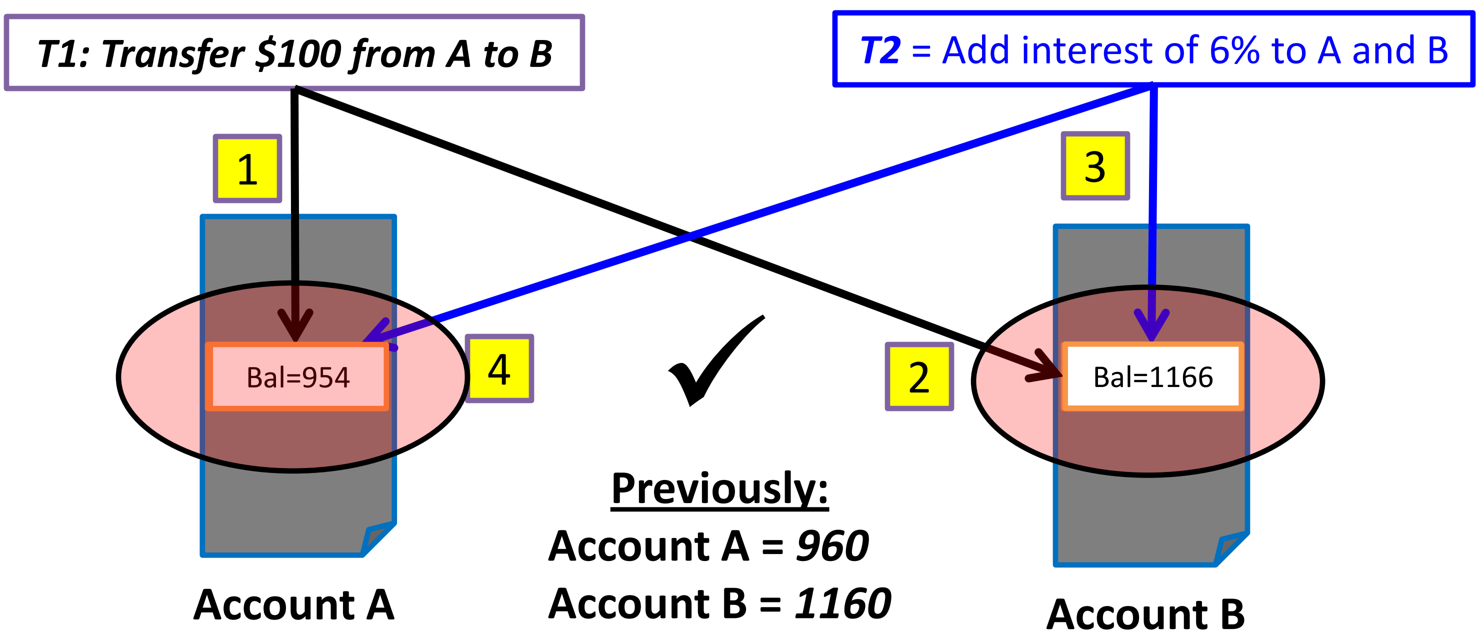
T1:	BEGIN	A=A-100,	B=B +100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END



Examples: Another *Serial* Schedule

- Assume transactions T1 and T2 as follows:

T1:	BEGIN	A=A-100,	B=B +100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

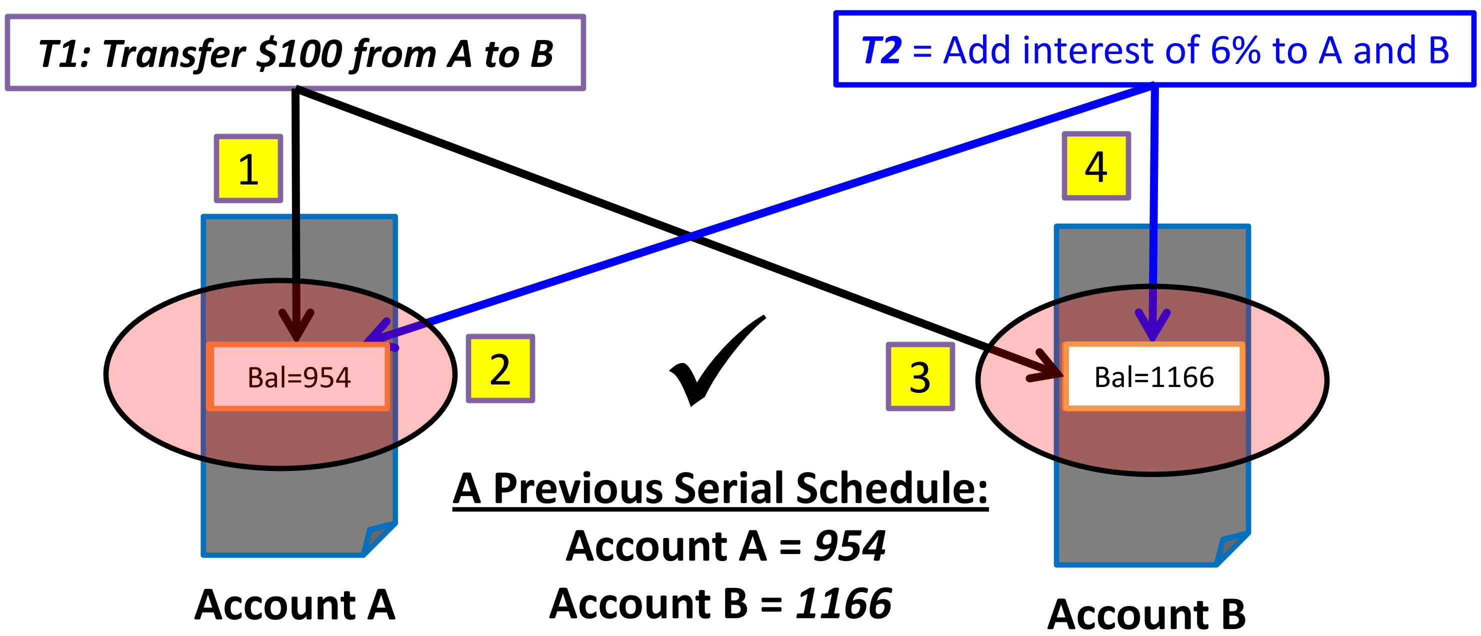


Previously:
Account A = *960*
Account B = *1160*

Examples: A *Serializable* Schedule

- Assume transactions T1 and T2 as follows:

T1:	BEGIN	A=A-100,	B=B +100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END



Comments

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together

- However, the net effect *must* be equivalent to these two transactions running *serially* in some order

- Executing transactions serially in different orders may produce different results, but they are all acceptable!

- The DBMS makes no guarantees about which result will be the outcome of an interleaved execution

Outline

- A Brief Primer on Transaction Management
- Anomalies Due to Concurrency ✓
- 2PL and Strict 2PL Locking Protocols
- Schedules with Aborted Transactions

Anomalies

- Interleaving actions of different transactions can leave the database in an inconsistent state

- Two actions on the same data object are said to ***conflict*** if at least one of them is a write

- There are 3 anomalies that can arise upon interleaving actions of different transactions (say, T1 and T2):
  - Write-Read (WR) Conflict: T2 reads a data object previously written by T1
  - Read-Write (RW) Conflict: T2 writes a data object previously read by T1
  - Write-Write (WW) Conflict: T2 writes a data object previously written by T1

Reading Uncommitted Data: WR Conflicts

- WR conflicts arise when transaction T2 reads a data object A that has been modified by another transaction T1, *which has not yet committed*
  - Such a read is called a dirty read

- Assume T1 and T2 such that:
  - T1 transfers $100 from A's account to B's account
  - T2 credits accounts A and B with a 6% interest payment

```
T1:    BEGIN   A=A-100,   B=B +100   END
T2:    BEGIN   A=1.06*A,   B=1.06*B   END
```

Reading Uncommitted Data: WR Conflicts

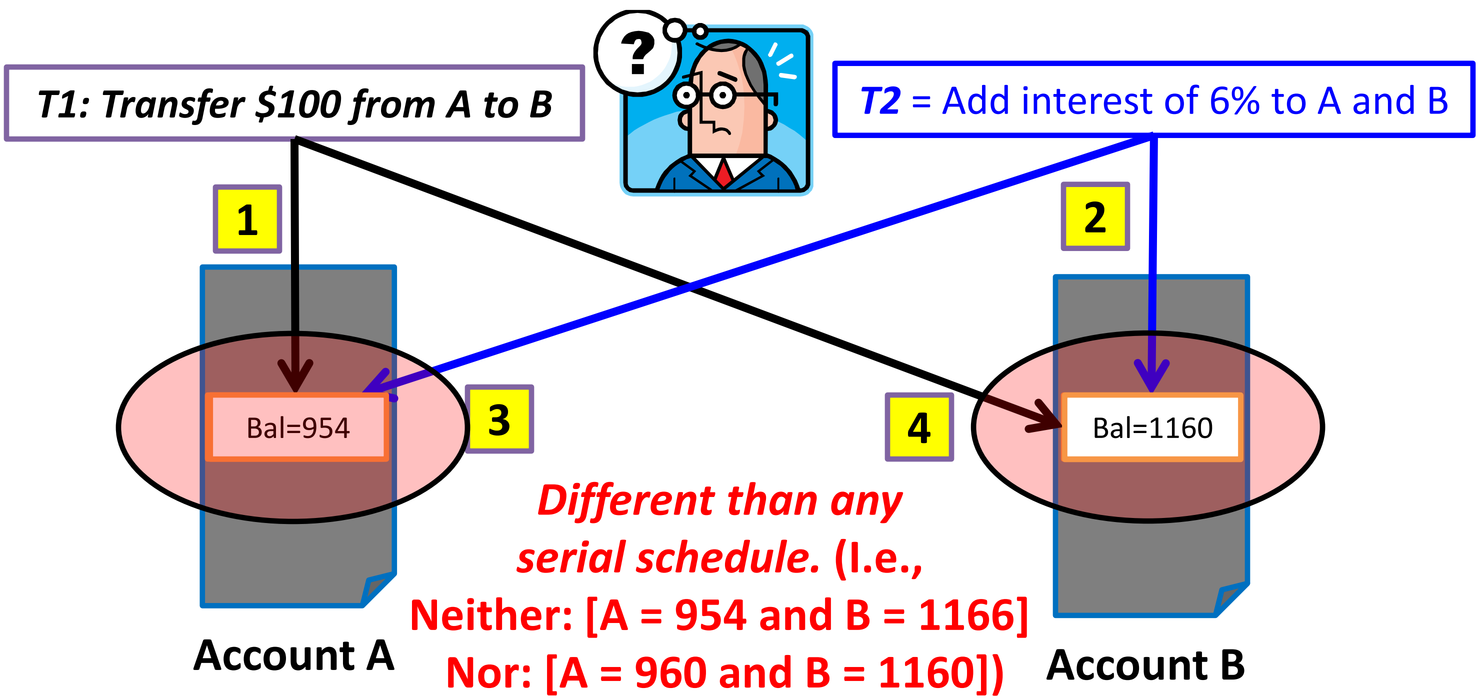
- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

T1: Transfer $100 from A to B

T2 = Add interest of 6% to A and B

Reading Uncommitted Data: WR Conflicts

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A **1**
  - T2 adds 6% interest to accounts A and B **2 and 3**
  - T1 credits $100 to account B **4**



Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:

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

The value of A written by T1 is read by T2 before T1 has completed all its changes!

Why is this a problem?

Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:

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

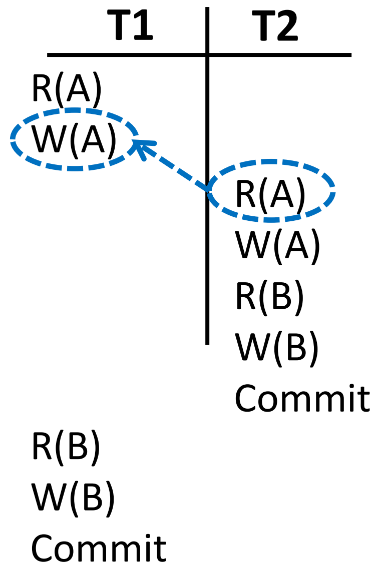
The value of A written by T1 is read by T2 before T1 has completed all its changes!

Why is this a problem?

- T1 may write some value into A that makes the database inconsistent
- As long as T1 overwrites this value with a 'correct' value of A before committing, no harm is done if T1 and T2 are run in some serial order (this is because T2 would then not see the <u>temporary</u> inconsistency)

Reading Uncommitted Data: WR Conflicts

- T1 and T2 can be represented by the following schedule:



The value of A written by T1 is read by T2 before T1 has completed all its changes!

Why is this a problem?

Note that although a transaction must leave a database in a consistent state *after* it completes, it is not required to keep the database consistent while it is still in progress!

Unrepeatable Reads: RW Conflicts

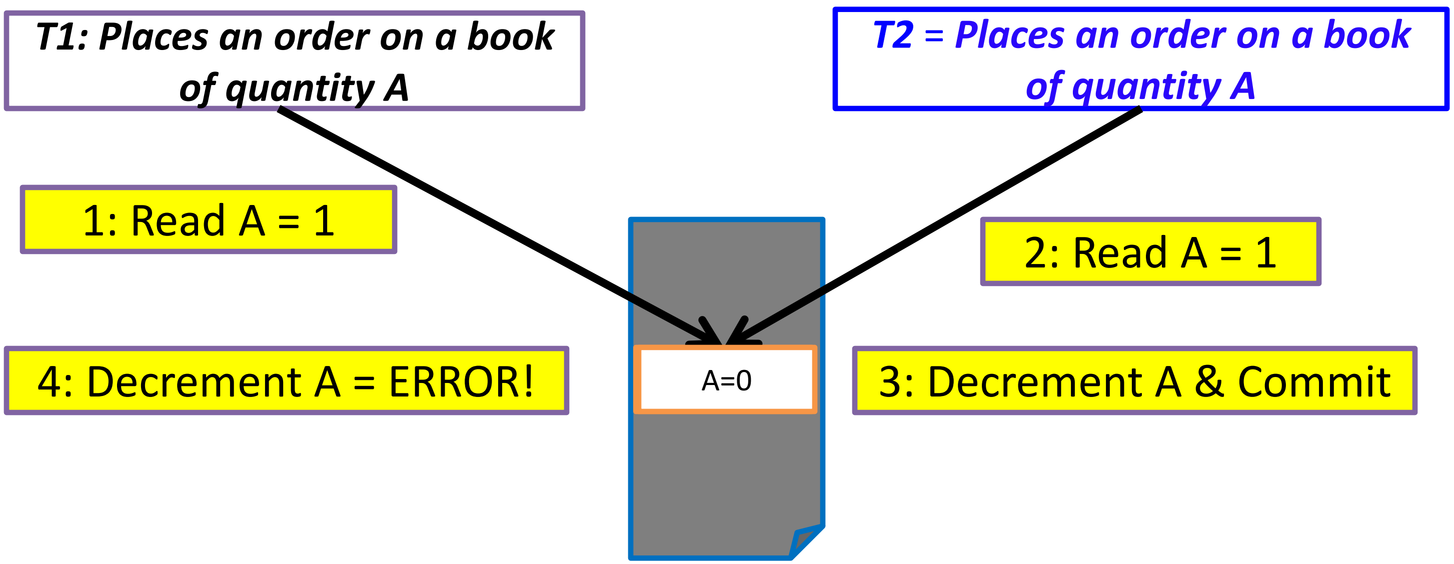
- RW conflicts arise when transaction T2 writes a data object A that has been read by another transaction T1, *while T1 is still in progress*

- If T1 tries to read A again, it will get a different result!
  - Such a read is called an unrepeatable read

- Assume A is the number of available copies for a book
  - A transaction that places an order on the book reads A, checks that A > 0 and decrements A
  - Assume two transactions, T1 and T2

Unrepeatable Reads: RW Conflicts

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 reads A
  - T2 reads A, decrements A and commit
  - T1 tries to decrement A



This situation will never arise in a serial execution of T1 and T2; T2 would read A and see 0 and therefore not proceed with placing an order!

Overwriting Uncommitted Data: WW Conflicts

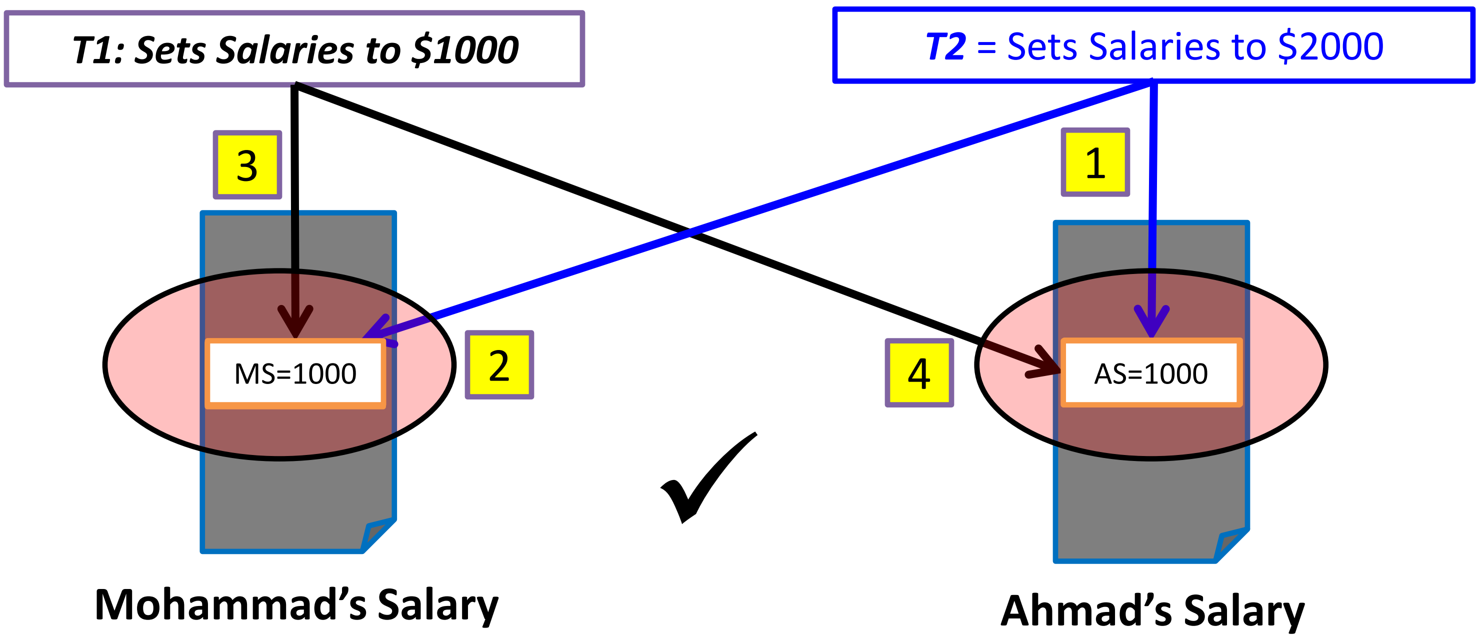
- WW conflicts arise when transaction T2 writes a data object A that has been written by another transaction T1, *while T1 is still in progress*

- Suppose that Mohammad and Ahmad are two employees and their salaries *must be kept equal*

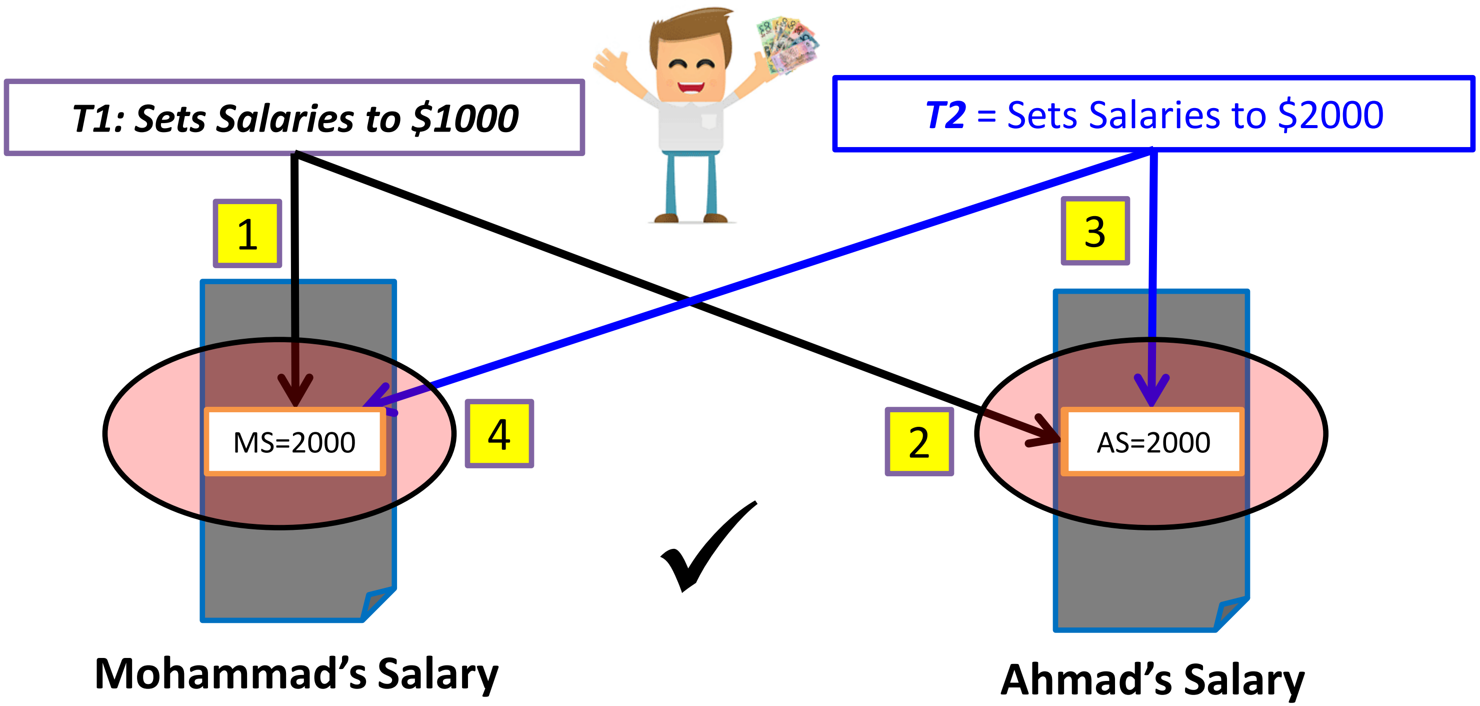
- Assume T1 sets Mohammad's and Ahmad's salaries to $1000

- Assume T2 sets Mohammad's and Ahmad's salaries to $2000

Overwriting Uncommitted Data: WW Conflicts



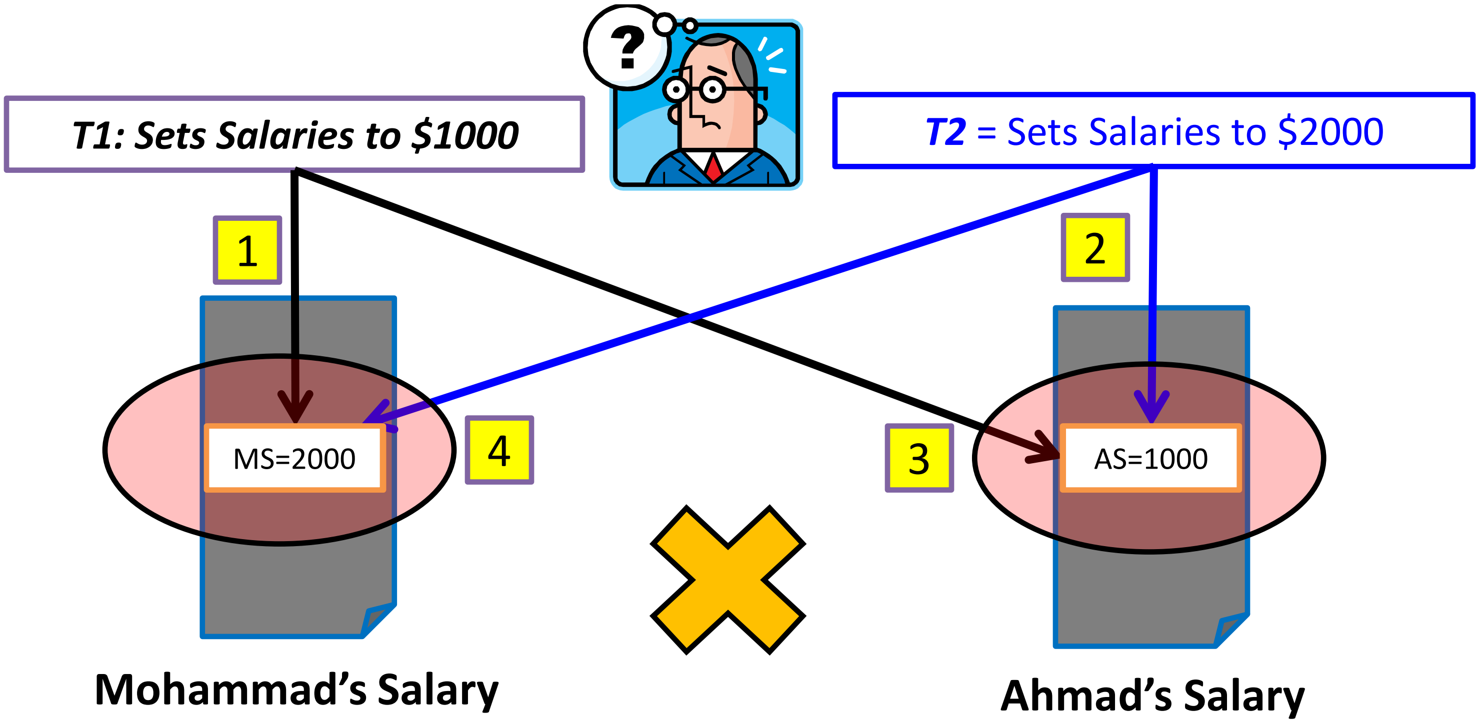
Overwriting Uncommitted Data: WW Conflicts



Either serial schedule is <u>acceptable</u> from a *consistency standpoint* (although Mohammad and Ahmad may prefer a higher salary!)

Neither T1 nor T2 reads a salary value before writing it- such a write is called a ***blind write!***

Overwriting Uncommitted Data: WW Conflicts



The problem is that we have a ***lost update***. In particular, T2 overwrote Mohammad's Salary as set by T1 (this will never happen with a serializable schedule!)

Outline

- A Brief Primer on Transaction Management
- Anomalies Due to Concurrency
- 2PL and Strict 2PL Locking Protocols ✓
- Schedules with Aborted Transactions

Locking Protocols

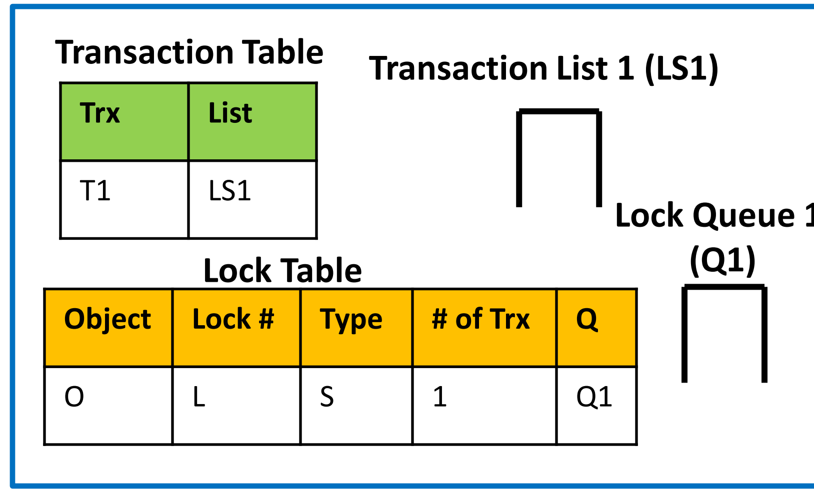
- WR, RW and WW anomalies can be avoided using a *locking protocol*

- A locking protocol:
  - Is *a set of rules* to be followed by each transaction to ensure that only serializable schedules are allowed (*extended later*)
  - Associates a *lock* with each database object, which could be of different types (e.g., *shared* or *exclusive*)
  - *Grants and denies locks* to transactions according to the specified rules

- The part of the DBMS that keeps track of locks is called the *lock manager*

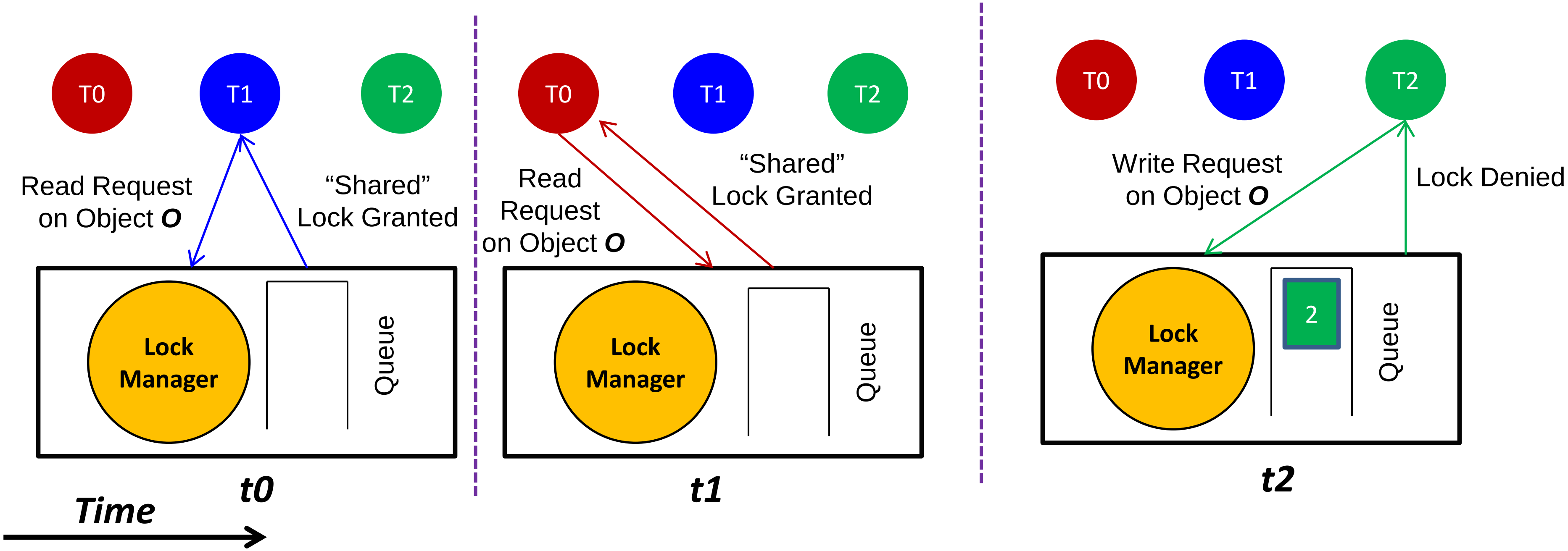
Lock Managers

- Usually, a lock manager in a DBMS maintains three types of data structures:
  - A queue, ***Q***, for each lock, ***L***, to hold its pending requests
  - A lock table, which keeps for each ***L*** associated with each object, ***O***, a record ***R*** that contains:
    - The type of ***L*** (e.g., shared or exclusive)
    - The number of transactions currently holding ***L*** on ***O***
    - A pointer to ***Q***
  - A transaction table, which maintains for each transaction, ***T***, a pointer to a list of locks held by ***T***



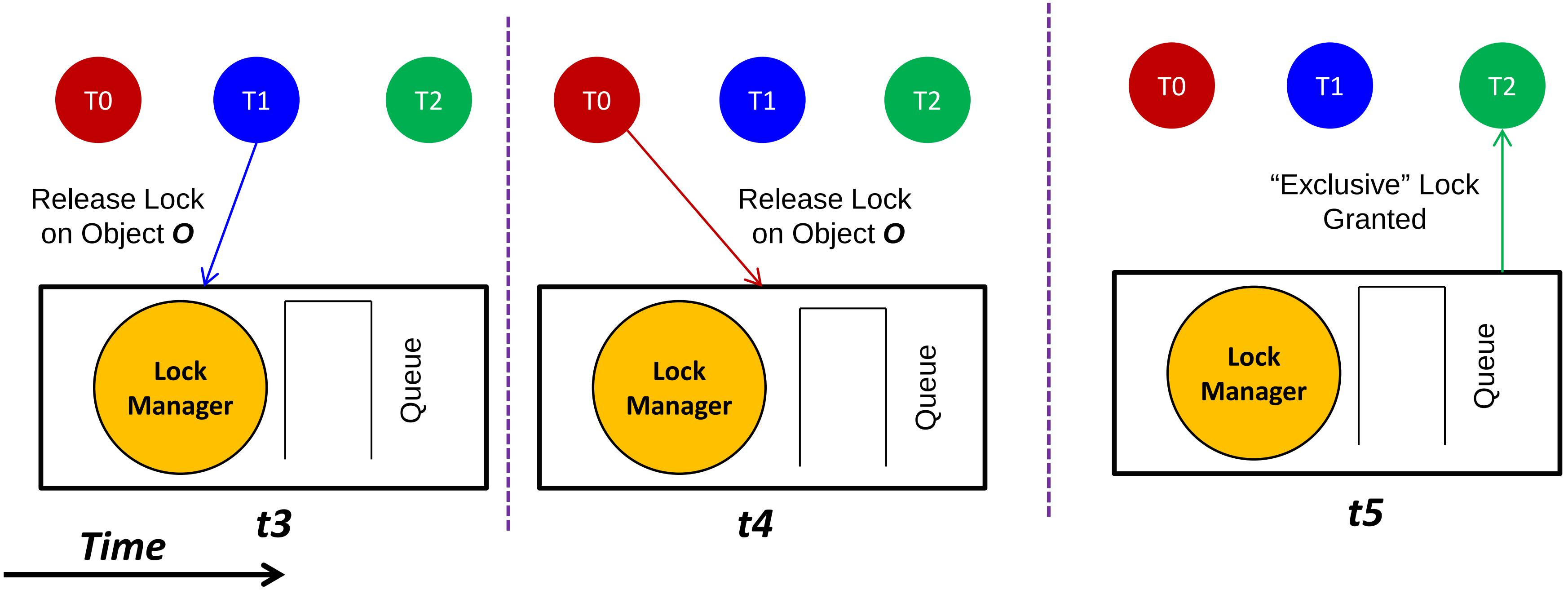
Two-Phase Locking

- A widely used locking protocol, called *Two-Phase Locking* (*2PL*), has two rules:
  - Rule 1: if a transaction ***T*** wants to read (or write) an object ***O***, it first requests the lock manager for a shared (or exclusive) lock on ***O***



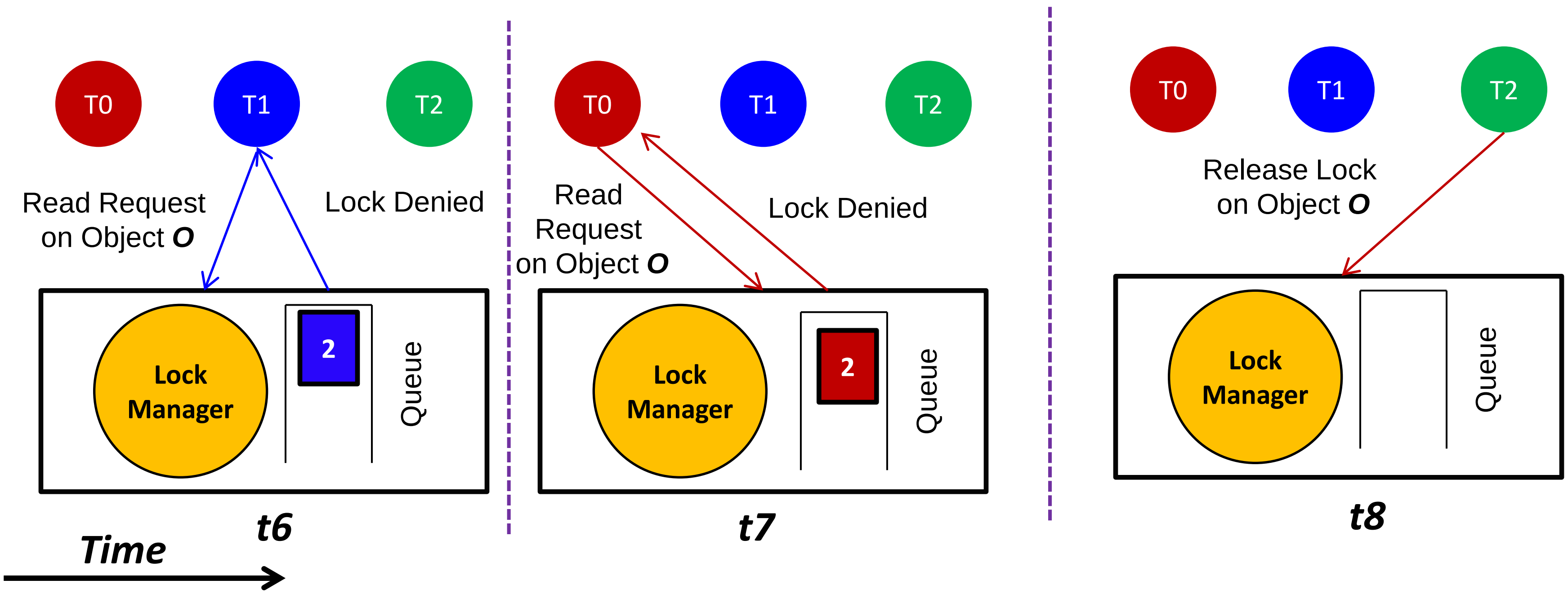
Two-Phase Locking

- A widely used locking protocol, called *Two-Phase Locking* (*2PL*), has two rules:
  - Rule 1: if a transaction ***T*** wants to read (or write) an object ***O***, it first requests the lock manager for a shared (or exclusive) lock on ***O***



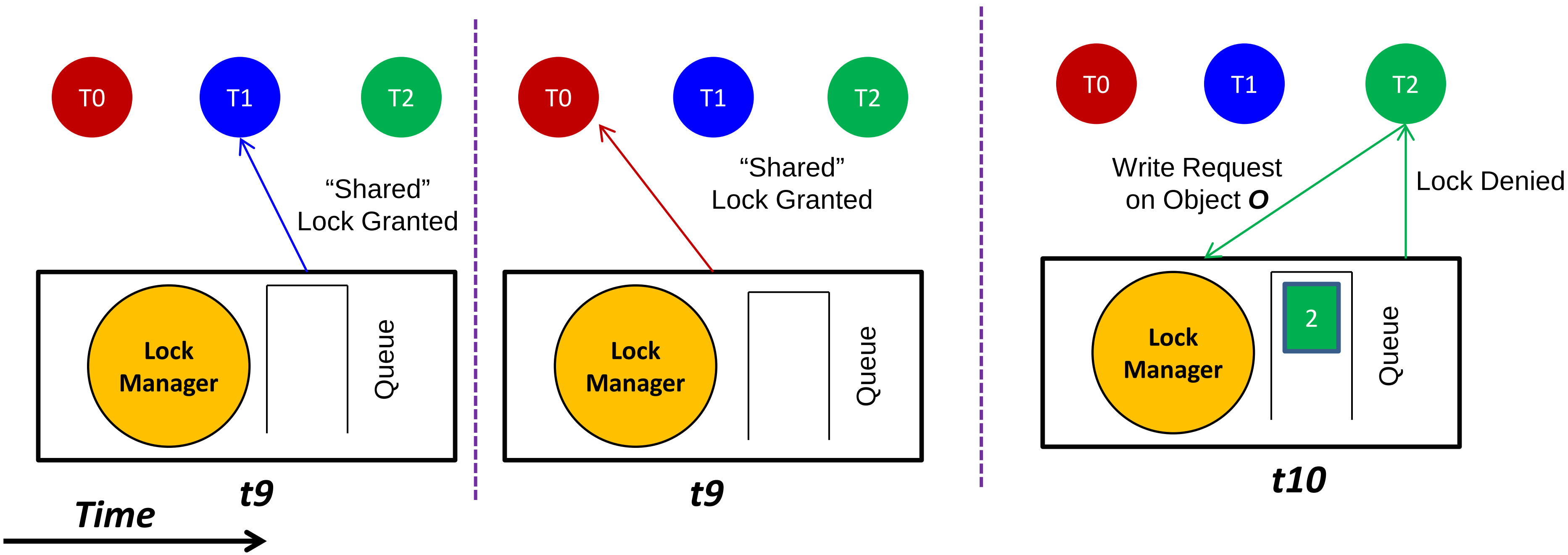
Two-Phase Locking

- A widely used locking protocol, called *Two-Phase Locking* (*2PL*), has two rules:
  - Rule 1: if a transaction ***T*** wants to read (or write) an object ***O***, it first requests the lock manager for a shared (or exclusive) lock on ***O***



Two-Phase Locking

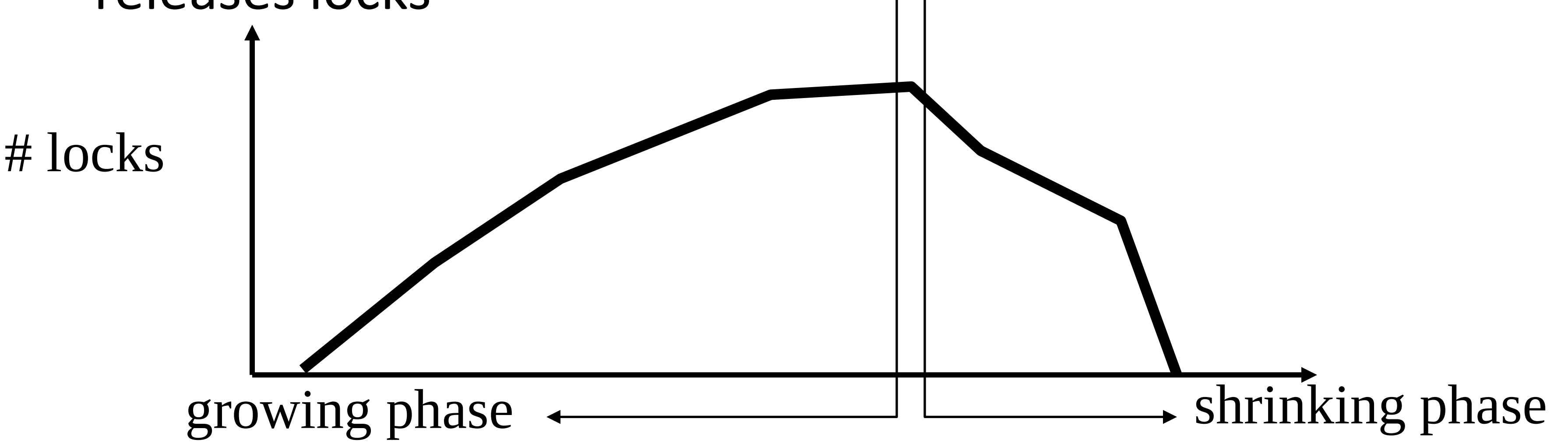
- A widely used locking protocol, called *Two-Phase Locking* (*2PL*), has two rules:
  - Rule 1: if a transaction ***T*** wants to read (or write) an object ***O***, it first requests the lock manager for a shared (or exclusive) lock on ***O***



Two-Phase Locking

- A widely used locking protocol, called *Two-Phase Locking* (*2PL*), has two rules:
  - Rule 2: ***T*** can release locks before it *commits* or *aborts*, and cannot request additional locks once it releases *any* lock

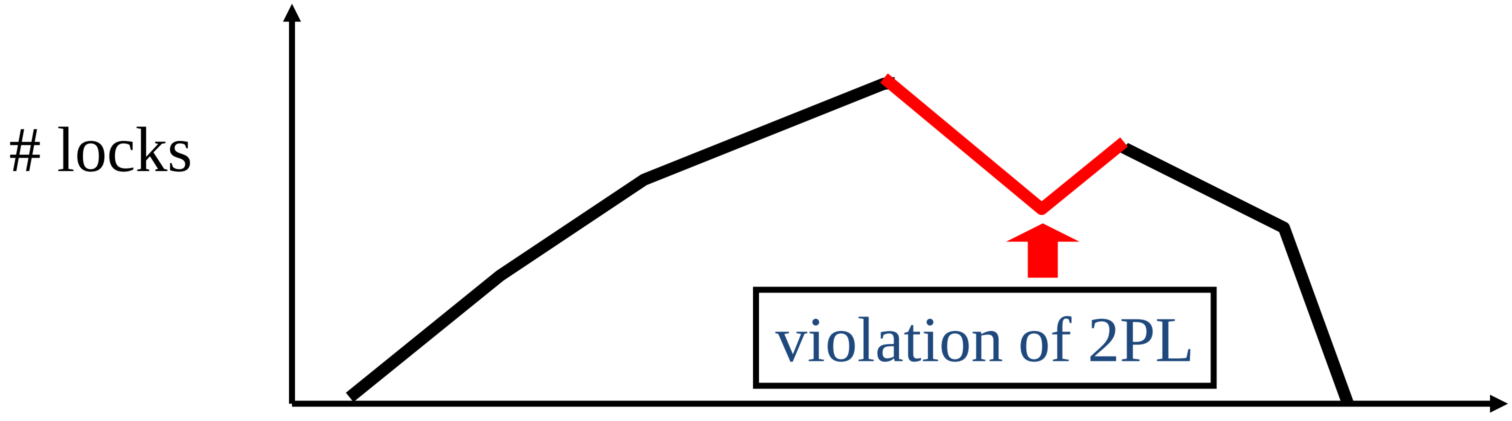
- Thus, every transaction has a "growing" phase in which it acquires locks, followed by a "shrinking" phase in which it releases locks



Two-Phase Locking

- A widely used locking protocol, called *Two-Phase Locking* (*2PL*), has two rules:
  - Rule 2: ***T*** can release locks before it *commits* or *aborts*, and cannot request additional locks once it releases *any* lock

- Thus, every transaction has a "growing" phase in which it acquires locks, followed by a "shrinking" phase in which it releases locks



Resolving RW Conflicts Using 2PL

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 reads A
  - T2 reads A, decrements A and commit
  - T1 tries to decrement A

- T1 and T2 can be represented by the following schedule:

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

Exposes RW Anomaly

T1	T2
EXCLUSIVE(A)	
R(A)	
	Lock(A)
W(A)	**Wait**
Commit	
	EXCLUSIVE(A)
	R(A)
	W(A)
	Commit

RW Conflict Resolved!

Resolving RW Conflicts Using 2PL

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 reads A
  - T2 reads A, decrements A and commit
  - T1 tries to decrement A
- T1 and T2 can be represented by the following schedule:

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

Exposes RW Anomaly

T1	T2
EXCLUSIVE(A)	
R(A)	
	Lock(A)
W(A)	**Wait**
Commit	
	EXCLUSIVE(A)
	R(A)
	W(A)
	Commit

But, it can limit parallelism!

Resolving WW Conflicts Using 2PL

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 sets Mohammad's Salary to $1000
  - T2 sets Ahmad's Salary to $2000
  - T1 sets Ahmad's Salary to $1000
  - T2 sets Mohammad's Salary to $2000
- T1 and T2 can be represented by the following schedule:

T1	T2
W(MS)	
	W(AS)
W(AS)	
Commit	W(MS)
	Commit

Exposes WW Anomaly
(*assuming, MS & AS must be kept equal*)

T1	T2
EXCLUSIVE(MS)	
EXCLUSIVE(AS)	
W(MS)	**Lock**(AS)
W(AS)	↑ **Wait** ↓
Commit	
	EXCLUSIVE(AS)
	EXCLUSIVE(MS)
	W(AS)
	W(MS)
	Commit

WW Conflict Resolved!

Resolving WW Conflicts Using 2PL

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 sets Mohammad's Salary to $1000
  - T2 sets Ahmad's Salary to $2000
  - T1 sets Ahmad's Salary to $1000
  - T2 sets Mohammad's Salary to $2000

- T1 and T2 can be represented by the following schedule:

T1	T2
W(MS)	
	W(AS)
W(AS)	
Commit	W(MS)
	Commit

Exposes WW Anomaly
(*assuming, MS & AS must be kept equal*)

→

T1	T2
EXCLUSIVE(MS)	
W(MS)	
	EXCLUSIVE(AS)
Lock(AS)	W(AS)
↑ **Wait**	**Lock**(MS)
	↑ **Wait**

Deadlock!

Resolving WR Conflicts

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

- T1 and T2 can be represented by the following schedule:

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

Exposes WR Anomaly

T1	T2
EXCLUSIVE(A)	
EXCLUSIVE(B)	**Lock**(A)
R(A)	**Lock**(B)
W(A)	
R(B)	**Wait**
W(B)	
Commit	
	EXCLUSIVE(A)
	EXCLUSIVE(B)
	R(A)
	W(A)
	R(B)
	W(B)
	Commit

WR Conflict Resolved!

Resolving WR Conflicts

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

- T1 and T2 can be represented by the following schedule:

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

Exposes WR Anomaly

T1	T2
EXCLUSIVE(A)	
EXCLUSIVE(B)	**Lock**(A)
R(A)	**Lock**(B)
W(A)	**Wait**
RELEASE(A)	↓
R(B)	**EXCLUSIVE**(A)
W(B)	R(A)
Commit	W(A)
	EXCLUSIVE(B)
	R(B)
	W(B)
	Commit

WR Conflict is ***NOT*** Resolved!

How can we solve this?

Strict Two-Phase Locking

- WR conflicts (as well as RW & WW) can be solved by making 2PL *stricter*

- In particular, *Rule 2* in 2PL can be modified as follows:
  - Rule 2: locks of a transaction ***T*** can only be released after ***T*** completes (i.e., commits or aborts)

- This version of 2PL is called *Strict* Two-Phase Locking

Resolving WR Conflicts: *Revisit*

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

- T1 and T2 can be represented by the following schedule:

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

Exposes WR Anomaly

T1	T2
EXCLUSIVE(A)	
EXCLUSIVE(B)	**Lock**(A)
R(A)	**Lock**(B)
W(A)	**Wait**
RELEASE(A)	↓
R(B)	**EXCLUSIVE**(A)
W(B)	R(A)
Commit	W(A)
	EXCLUSIVE(B)
	R(B)
	W(B)
	Commit

Not allowed with *strict* 2PL

Resolving WR Conflicts: *Revisit*

- Suppose that T1 and T2 actions are *interleaved* as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B
  - T1 credits $100 to account B

- T1 and T2 can be represented by the following schedule:

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

Exposes WR Anomaly

T1	T2
EXCLUSIVE(A)	
EXCLUSIVE(B)	
R(A)	**Lock**(A)
W(A)	**Lock**(B)
R(B)	↑
W(B)	**Wait**
Commit	↓
	EXCLUSIVE(A)
	EXCLUSIVE(B)
	R(A)
	W(A)
	R(B)
	W(B)
	Commit

WR Conflict is Resolved!

But, parallelism is limited more!

2PL vs. Strict 2PL

- Two-Phase Locking (2PL):
  - Limits concurrency
  - May lead to deadlocks
  - May have ‘dirty reads’

- Strict 2PL:
  - Limits concurrency more (*but*, actions of different transactions can still be interleaved)
  - May still lead to deadlocks
  - Avoids ‘dirty reads’

T1	T2
SHARED(A)	
R(A)	
	SHARED(A)
	R(A)
	EXECLUSIVE(B)
	R(B)
EXCLUSIVE(C)	W(B)
R(C)	Commit
W(C)	
Commit	

A Schedule with *Strict 2PL* and *Interleaved Actions*

Performance of Locking

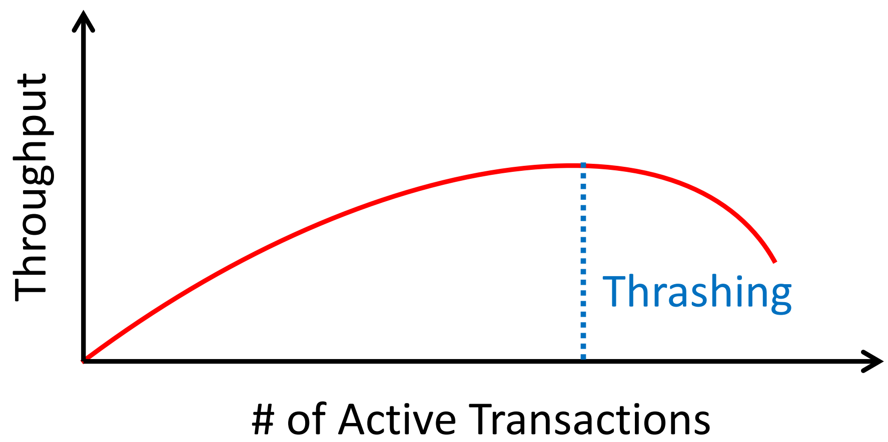
- Locking comes with delays mainly from *blocking*

- Usually, the first few transactions are unlikely to conflict
  - Throughput can rise in proportion to the number of active transactions

- As more transactions are executed concurrently, the likelihood of blocking increases
  - Throughput will increase more slowly with the number of active transactions

- There comes a point when adding another active transaction will actually decrease throughput
  - When the system *thrashes*!

Performance of Locking (*Cont'd*)



- If a database begins to *thrash*, the DBA should reduce the number of active transactions

- Empirically, thrashing is seen to occur when 30% of active transactions are blocked!

Outline

- A Brief Primer on Transaction Management
- Anomalies Due to Concurrency
- 2PL and Strict 2PL Locking Protocols
- Schedules with Aborted Transactions ✓

Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

T2 read a value for A that should never have been there!

How can we deal with the situation, assuming T2 *had not yet committed*?

Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

T2 read a value for A that should never have been there!

We can ***cascade*** the abort of T1 by aborting T2 as well!

This “cascading process” can be *recursively* applied to any transaction that read A written by T1

Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

T2 read a value for A that should never have been there!

How can we deal with the situation, assuming T2 *had actually committed*?

The schedule is indeed *unrecoverable*!

Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

T2 read a value for A that should never have been there!

For a schedule to be *recoverable*, transactions should commit only after all transactions whose changes they read commit!

"Recoverable schedules" avoid *cascading aborts*!

Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

T2 read a value for A that should never have been there!

How can we ensure "recoverable schedules"?

By using Strict 2PL!

Schedules with *Aborted* Transactions

- Suppose that T1 and T2 actions are interleaved as follows:
  - T1 deducts $100 from account A
  - T2 adds 6% interest to accounts A and B, and commits
  - T1 is aborted

- T1 and T2 can be represented by the following schedule:

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

T1	T2
EXCLUSIVE(A)	
R(A)	
W(A)	**Lock**(A)
	Wait ↓
Abort	
UNDO(T1)	**EXCLUSIVE**(A)
	R(A)
	W(A)
	EXCLUSIVE(B)
	R(B)
	W(B)
	Commit

Cascaded aborts are avoided!

Serializable Schedules: *Redefined*

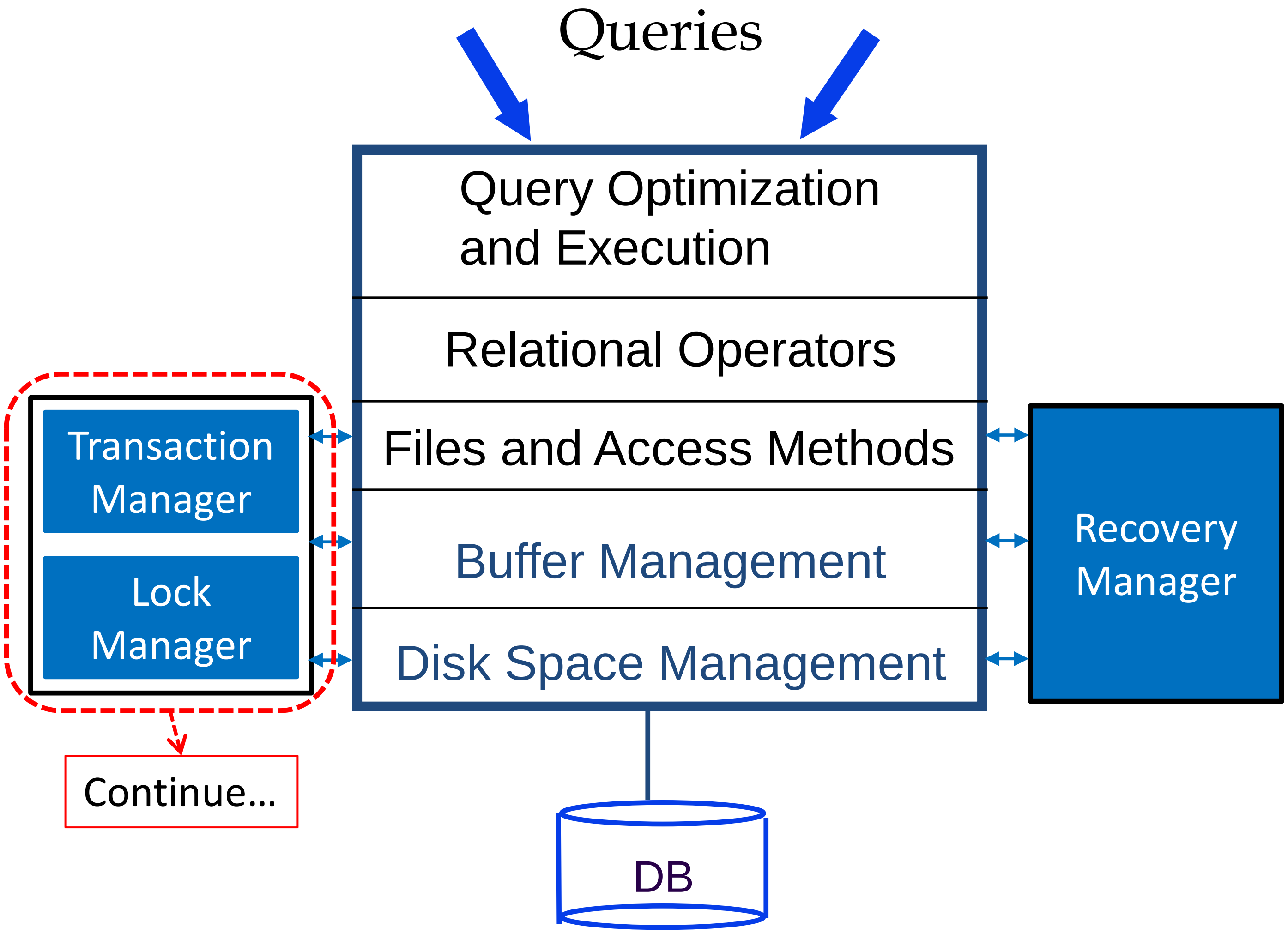
- Two schedules are said to be *equivalent* if for any database state, the effect of executing the 1st schedule is identical to the effect of executing the 2nd schedule

- Previously: a *serializable schedule* is a schedule that is equivalent to a serial schedule

- Now: a *serializable schedule* is a schedule that is equivalent to a serial schedule *over a set of committed transactions*

- This definition captures *serializability* as well as *recoverability*

Next Class


Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction
Manager
Lock
Manager
Continue…
Recovery
Manager
DB