Transactions & Concurrency Control

Bassam Hammo

Transactions

- A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.

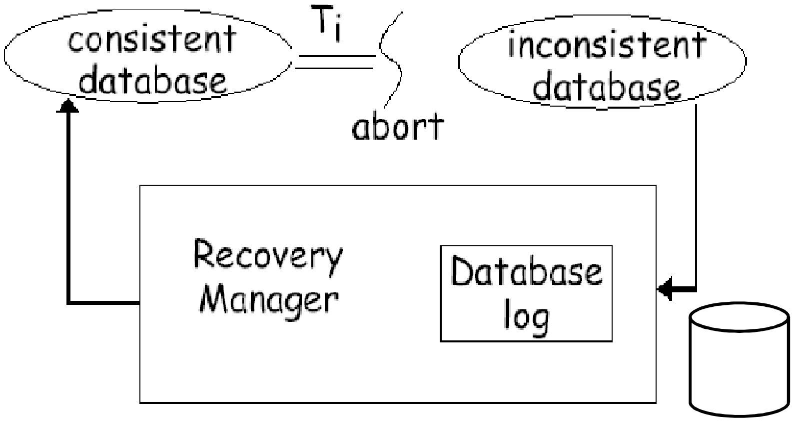
Transactions

- A transaction is a 'logical unit of work' on a database
  - Each transaction does something in the database
  - No part of it alone achieves anything of use or interest
- Transactions are the unit of recovery, consistency, and integrity as well
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity and Consistency

- Atomicity
  - Transactions are atomic – they don't have parts (conceptually)
  - can't be executed partially; it should not be detectable that they interleave with another transaction
- Consistency
  - Transactions take the database from one consistent state into another
  - In the middle of a transaction the database might not be consistent

Atomicity


consistent database

T_i

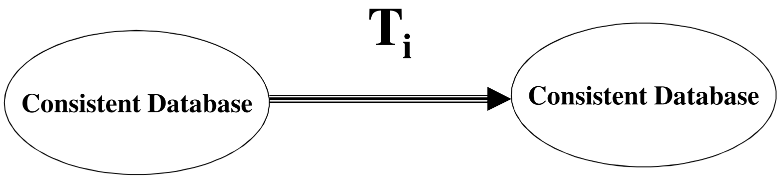
abort

inconsistent database

Recovery Manager

Database log

Consistency


T_i

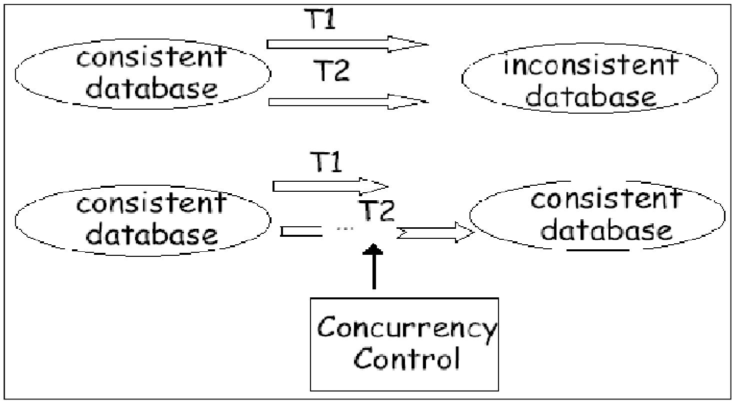
Consistent Database

Consistent Database

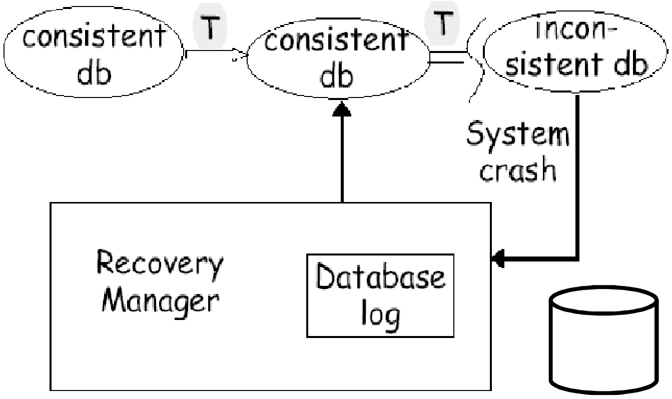
Isolation and Durability

- Isolation
  - The effects of a transaction are not visible to other transactions until it has completed
  - From outside the transaction has either happened or not
  - To me this actually sounds like a consequence of atomicity…
- Durability
  - Once a transaction has completed, its changes are made permanent
  - Even if the system crashes, the effects of a transaction must remain in place

Isolation



Global Recovery



Example of a transaction

- Transfer 50 JD from account A to account B

  Read(A)
  A = A - 50
  Write(A)
  Read(B)
  B = B+50
  Write(B)

  transaction

Atomicity - shouldn't take money from A without giving it to B

Consistency - money isn't lost or gained

Isolation - other queries shouldn't see A or B change until completion

Durability - the money does not go back to A

The Transaction Manager

- The transaction manager enforces the ACID properties
  - It schedules the operations of transactions
  - COMMIT and ROLLBACK are used to ensure atomicity
- Locks or timestamps are used to ensure consistency and isolation for concurrent transactions (next lectures)
- A log is kept to ensure durability in the event of system failure (discussed)

Concurrency

- Large databases are used by many people
  - Many transactions to be run on the database
  - It is desirable to let them run at the same time as each other
  - Need to preserve isolation
- If we don't allow for concurrency then transactions are run sequentially
  - Have a queue of transactions
  - Long transactions (e.g. backups) will make others wait for long periods

Concurrency Problems

- In order to run transactions concurrently we interleave their operations
- Each transaction gets a share of the computing time
- This leads to several sorts of problems
  - **Lost updates**
  - **Uncommitted updates**
  - **Incorrect analysis**
- All arise because isolation is broken

Lost Update

- T1 and T2 read X, both modify it, then both write it out
  - The net effect of T1 and T2 should be no change on X
  - Only T2's change is seen, however, so the final value of X has increased by 5

T1	**T2**
`Read(X)`	
`X = X - 5`	
	`Read(X)`
	`X = X + 5`
`Write(X)`	
	`Write(X)`
`COMMIT`	
	`COMMIT`

Uncommitted Update

- T2 sees the change to X made by T1, but T1 is rolled back
  - The change made by T1 is undone on rollback
  - It should be as if that change never happened

T1	T2
`Read(X)`	
`X = X - 5`	
`Write(X)`	
	`Read(X)`
	`X = X + 5`
	`Write(X)`
`ROLLBACK`	
	`COMMIT`

Inconsistent analysis

- T1 doesn't change the sum of X and Y, but T2 sees a change
  - T1 consists of two parts – take 5 from X and then add 5 to Y
  - T2 sees the effect of the first, but not the second

T1	T2
`Read(X)`	
`X = X - 5`	
`Write(X)`	
	`Read(X)`
	`Read(Y)`
	`Sum = X+Y`
`Read(Y)`	
`Y = Y + 5`	
`Write(Y)`	

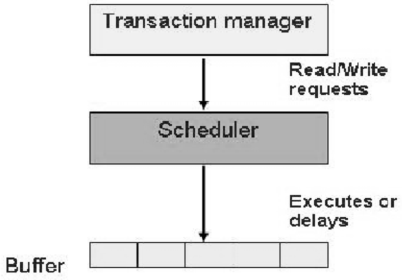
Need for concurrency control

- Transactions running concurrently may interfere with each other, causing various problems (lost updates etc.)
- Concurrency control: the process of managing simultaneous operations on the database without having them interfere with each other.

Schedules

- A *schedule* is a sequence of the operations by a set of concurrent transactions that preserves the order of operations in each of the individual transactions
- A *serial* schedule is a schedule where operations of each transaction are executed consecutively without any interleaved operations from other transactions (each transaction commits before the next one is allowed to begin)

The Scheduler



- The scheduler component of a DBMS must ensure that the individual steps of different transactions preserve consistency.

Serial schedules

- Serial schedules are guaranteed to avoid interference and keep the database consistent
- However databases need concurrent access which means interleaving operations from different transactions

Serializability

- The objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another.
- In other words, we want to find nonserial schedules that are equivalent to some serial schedule. Such a schedule is called serializable.

Uses of Serializability

- being serializable means
  - the schedule is **equivalent** to some serial schedule
  - Serial schedules are correct
  - Therefore, serializable schedules are also correct schedules
- serializability is hard to test
  - Use precedence graph (PG)
- Need the methods (or protocols) to enforce serializabilty
  - Two phase locking(2PL)
  - Time stamp ordering (TSO)

Conflict Serialisability

- Conflict serialisable schedules are the main focus of concurrency control
- They allow for interleaving and at the same time they are guaranteed to behave as a serial schedule
- Important questions: how to determine whether a schedule is conflict serialisable
- How to construct conflict serialisable schedules

Conflicting Operations

No.	Case	Conflict	Non-Conf
1	I_i & I_j operate on different data items		X
2	I_i = Read(Q) & I_j = Read (Q)		X
3	I_i = Read(Q) & I_j = Write (Q)	X	
4	I_i = Write(Q) & I_j = Write (Q)	X	
5	I_i = Write(Q) & I_j = Read (Q)	X	

- **The only conflicting operation is the Write operation**

Precedence Graph (PG)

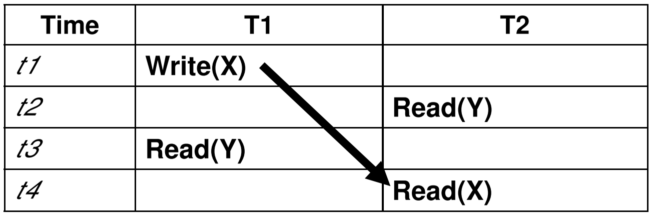
- Precedence graph
  - Used to test for conflict serializability of a schedule
  - A directed graph G=(V,E)
    - V: a finite set of transactions
    - E: a set of arcs from T_i to T_j if an action of T_i comes first and conflicts with one of T_j's actions

More on PG

- The serialization order is obtained through **topological sorting**
- A schedule **S** is conflict serializable iff there is no cycle in the precedence graph (**acyclic**)

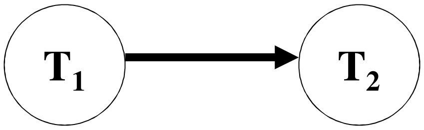
Serialization Graph

- Consider the schedule S:



Time	T1	T2
t1	**Write(X)**	
t2		**Read(Y)**
t3	**Read(Y)**	
t4		**Read(X)**

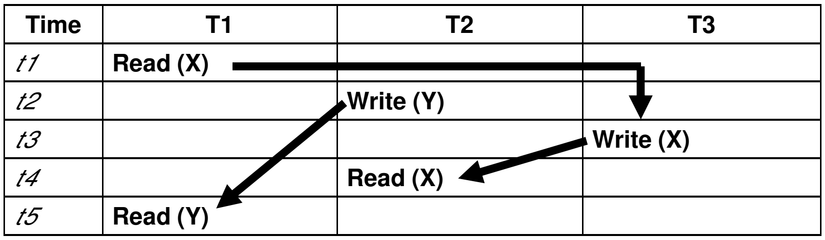
The precedence graph is



Thus, it is conflict equivalent to T_1,T_2

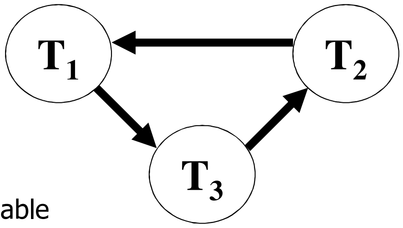
Serialization Graph

- Consider the schedule:



Time	T1	T2	T3
t1	**Read (X)**		
t2		**Write (Y)**	
t3			**Write (X)**
t4		**Read (X)**	
t5	**Read (Y)**		

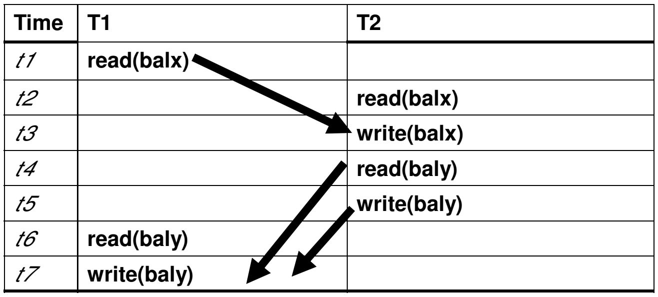
The precedence graph is



There is a cycle.
Hence it is NOT conflict serializable

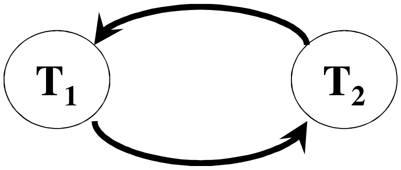
Serialization Graph

- Consider the schedule:



Time	T1	T2
t1	**read(balx)**	
t2		**read(balx)**
t3		**write(balx)**
t4		**read(baly)**
t5		**write(baly)**
t6	**read(baly)**	
t7	**write(baly)**	

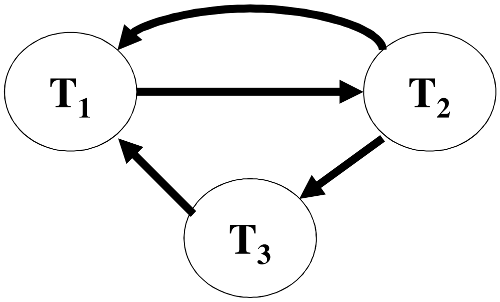
The precedence graph is



There is a cycle.
Hence it is NOT conflict serializable

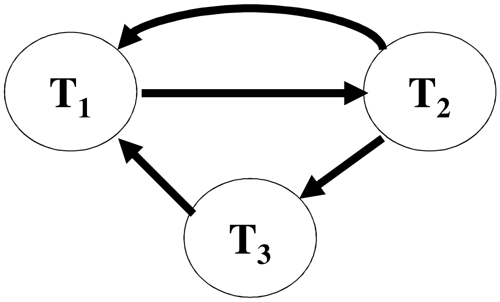
Serialization Graph

- Consider the following PG:



Serialization Graph

- Consider the following PG:



Cycle T1 ➔ T2 ➔ T1

Cycle T1 ➔ T2 ➔ T3 ➔ T1

Concurrency Control Techniques

- How can the DBMS ensure serializability?
- Two basic concurrency control techniques:
  - **Locking methods**
  - **Timestamping**

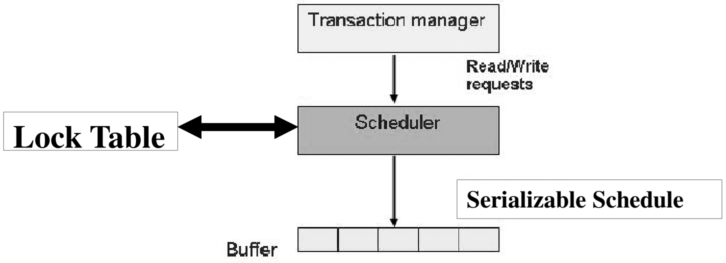
Locking

- Transaction uses locks to deny access to other transactions and so prevent incorrect updates.
- Generally, a transaction must claim a
  - read (shared), or
  - write (exclusive)

lock on a data item before read or write.

- Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Locking



Two-Phase Locking Protocol

- Each transaction issues lock and unlock requests in 2 phases:
  - **Growing phase**
    - A transaction may obtain locks, but may not release any lock
  - **Shrinking phase**
    - A transaction may release locks, but may not obtain any new locks

2 PL Protocol

- **Basics of locking:**

✓ Each transaction T must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.

✓ If an X lock is granted on object **O**, no other lock (X or S) might be granted on **O** at the same time.

✓ If an S lock is granted on object **O**, no X lock might be granted on **O** at the same time.

✓ Conflicting locks are expressed by the compatibility matrix:

	S	X
S	√	--
X	--	--

Basics of Locking

- A transaction does not request the same lock twice.
- A transaction does not need to request a S lock on an object for which it already holds an X lock.
- If a transaction has an S lock and needs an X lock it must wait until all other S locks (except its own) are released
- After a transaction has released one of its lock (unlock) it may not request any further locks (2PL: growing phase / shrinking phase)
- Using strict two-phase locking (strict 2PL) a transactions releases all its lock at the end of its execution.

(strict) 2PL allows only serializable schedules.

Preventing Lost Update Problem Using 2PL

Time	**T1**	**T2**
t1		**start**
t2	**start**	**lock-X**(balx)
t3	**lock-X**(balx)	read(balx)
t4	**wait**	balx=balx + 100
t5	**wait**	write(balx)
t6	**wait**	**commit**/**unlock(balx)**
t7	read(balx)	
t8	balx=balx -10	
t9	write(balx)	
t10	**commit**/**unlock(balx)**	

Preventing Uncommitted Dependency Problem using 2PL

Time	T1	T2
t1		**start**
t2		**lock-X**(balx)
t3		read(balx)
t4	**start**	balx=balx + 100
t5	**lock-X**(balx)	write(balx)
t6	**wait**	**rollback**/**unlock(balx)**
t7	read(balx)	
t8	balx=balx -10	
t9	write(balx)	
t10	**commit**/**unlock(balx)**	

Preventing Inconsistent Analysis Problem using 2PL

Time	T1	T2
t1		start
t2	start	sum=0
t3	lock-X(balx)	
t4	read(balx)	lock-S(balx)
t5	balx=balx -10	wait
t6	write (balx)	wait
t7	lock-X(balz)	wait
t8	read(balz)	wait
t9	balz=balz+10	wait
t10	write(balz)	wait
t11	commit/unlock(balx,balz)	wait
t12		read(balx)
t13		sum=sum+balx
t14		lock-S(baly)
t15		read(baly)
t16		sum=sum+baly
t17		lock-S(balz)
t18		read (balz)
t19		sum=sum+balz
t20		commit/unlock(balx,baly,balz)

Locking methods: problems

- **Deadlock:** May result when two (or more) transactions are each waiting for locks held by the other to be released.

Deadlock

consider the following partial schedule:

Time	T1	T2
t1	**lock-S**(A)	
t2		**lock-S**(B)
t3		read(B)
t4	read(A)	
t5	**lock-X**(B)	
t6		**lock-X**(A)

The transactions are now deadlocked

Deadlock Example

Time	T1	T2
t1	start	
t2	lock-X(balx)	start
t3	read(balx)	lock-X(baly)
t4	balx=balx -10	read(baly)
t5	write (balx)	baly=baly + 100
t6	lock-X(baly)	write (baly)
t7	wait	lock-X(balx)
t8	wait	wait
t9	wait	wait
t10	..	..

Deadlock Detection

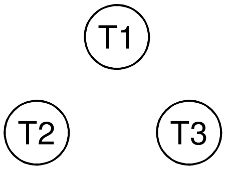
- Given a schedule, we can detect deadlocks which will happen in this schedule using a *wait-for graph* (WFG).

Precedence/Wait-For Graphs

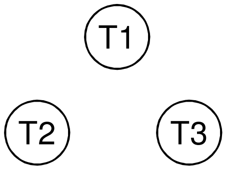
- Precedence graph
  - Each transaction is a vertex
  - Arcs from T1 to T2 if
    - T1 reads X before T2 writes X
    - T1 writes X before T2 reads X
    - T1 writes X before T2 writes X
- Wait-for Graph
  - Each transaction is a vertex
  - Arcs from T2 to T1 if
    - T1 read-locks X then T2 tries to write-lock it
    - T1 write-locks X then T2 tries to read-lock it
    - T1 write-locks X then T2 tries to write-lock it

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



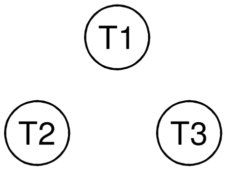
Wait for graph



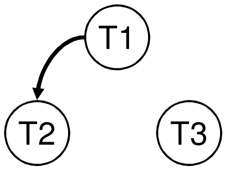
Precedence graph

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



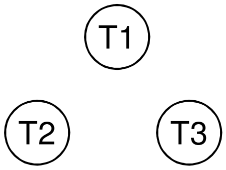
Wait for graph



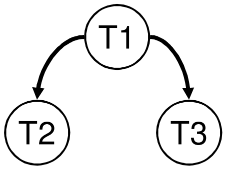
Precedence graph

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



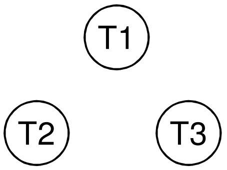
Wait for graph



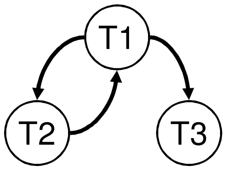
Precedence graph

Example

T1 Read(X)
T2 Read(Y)
T1 Write(X)
T2 Read(X)
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



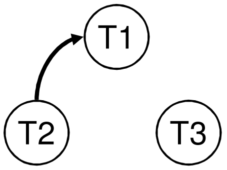
Wait for graph



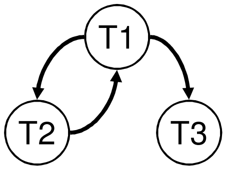
Precedence graph

Example

T1 Read(X) **S-lock(X)**
T2 Read(Y) **S-lock(Y)**
T1 Write(X) **X-lock(X)**
T2 Read(X) **tries S-lock(X)**
T3 Read(Z)
T3 Write(Z)
T1 Read(Y)
T3 Read(X)
T1 Write(Y)



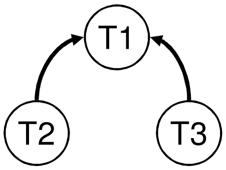
Wait for graph



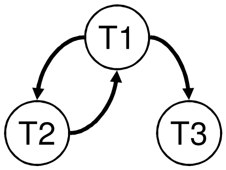
Precedence graph

Example

T1 Read(X) S-lock(X)
T2 Read(Y) **S-lock(Y)**
T1 Write(X) **X-lock(X)**
T2 Read(X) tries S-lock(X)
T3 Read(Z) **S-lock(Z)**
T3 Write(Z) **X-lock(Z)**
T1 Read(Y) **S-lock(Y)**
T3 Read(X) **tries S-lock(X)**
T1 Write(Y)



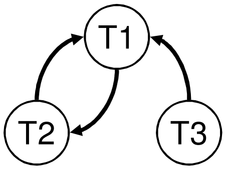
Wait for graph



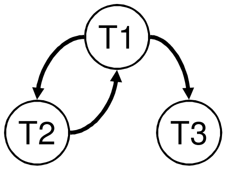
Precedence graph

Example

T1 Read(X) S-lock(X)
T2 Read(Y) **S-lock(Y)**
T1 Write(X) **X-lock(X)**
T2 Read(X) tries S-lock(X)
T3 Read(Z) **S-lock(Z)**
T3 Write(Z) **X-lock(Z)**
T1 Read(Y) **S-lock(Y)**
T3 Read(X) tries S-lock(X)
T1 Write(Y) **tries X-lock(Y)**



Wait for graph



Precedence graph

Solution

- Only one way to break deadlock: abort one or more of the transactions.
- Deadlock should be transparent to user, so DBMS should restart transaction(s).

Deadlock Prevention

- Deadlocks can arise with 2PL
  - Deadlock is less of a problem than an inconsistent DB
  - We can detect and recover from deadlock
  - It would be nice to avoid it altogether
- Conservative 2PL
  - All locks must be acquired before the transaction starts
  - Hard to predict what locks are needed
  - Low 'lock utilisation' - transactions can hold on to locks for a long time, but not use them much

Deadlock Prevention

- We impose an ordering on the resources
  - Transactions must acquire locks in this order
  - Transactions can be ordered on the last resource they locked
- This prevents deadlock
  - If T1 is waiting for a resource from T2 then that resource must come after all of T1's current locks
  - All the arcs in the wait-for graph point 'forwards' - no cycles

Example of resource ordering

- Suppose resource order is: X < Y
- This means, if you need locks on X and Y, you first acquire a lock on X and only after that a lock on Y
  - (even if you want to write to Y before doing anything to X)
- It is impossible to end up in a situation when T1 is waiting for a lock on X held by T2, and T2 is waiting for a lock on Y held by T1.

Timestamp

- Transactions can be run concurrently using a variety of techniques
- We looked at using locks to prevent interference

- An alternative is timestamping
  - Requires less overhead in terms of tracking locks or detecting deadlock
  - Determines the order of transactions before they are executed

---

Timestamp

- Each transaction has a timestamp, TS, and if T1 starts before T2 then TS(T1) < TS(T2)
  - Can use the system clock or an incrementing counter to generate timestamps

- Each resource has two timestamps
  - R(X), the largest timestamp of any transaction that has read X
  - W(X), the largest timestamp of any transaction that has written X

Timestamp Protocol

- If T tries to read X
  - If TS(T) < W(X) T is rolled back and restarted with a later timestamp
  - If TS(T) ≥ W(X) then the read succeeds and we set R(X) to be max(R(X), TS(T))
- T tries to write X
  - If TS(T) < W(X) or TS(T) < R(X) then T is rolled back and restarted with a later timestamp
  - Otherwise the write succeeds and we set W(X) to TS(T)

Timestamp Example 1

- Given T1 and T2 we will assume
  - The transactions make alternate operations
  - Timestamps are allocated from a counter starting at 1
  - T1 goes first

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

Timestamp Example 1

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R			
W			

	T1	T2
TS		

Timestamp Example 1

T1	T2
→ Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	1		
W			

	T1	T2
TS	1	

Timestamp Example 1

T1	T2
→ Read(X)	→ Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2		
W			

	T1	T2
TS	1	2

Timestamp Example 1

T1	T2
Read(X)	→ Read(X)
→ Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	1	
W			

	T1	T2
TS	1	2

Timestamp Example 1

T1	T2
Read(X)	Read(X)
→ Read(Y)	→ Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	1	2

Timestamp Example 1

T1	T2
Read(X)	Read(X)
Read(Y)	→ Read(Y)
→ Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	1	2

Timestamp Example 1

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
→ Y = Y + X	→ Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	1	2

Timestamp Example 1

T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	→ Z = Y - X
→ Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	1	2

Timestamp Example 1

→ T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	→ Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	2	2	
W			

	T1	T2
TS	3	2

Timestamp Example 1

→ T1	T2
Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	→ Write(Z)

	X	Y	Z
R	2	2	
W			2

	T1	T2
TS	3	2

Timestamp Example 1

T1	T2
→ Read(X)	Read(X)
Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	3	2	
W			2

	T1	T2
TS	3	2

Timestamp Example 1

T1	T2
Read(X)	Read(X)
→ Read(Y)	Read(Y)
Y = Y + X	Z = Y - X
Write(Y)	Write(Z)

	X	Y	Z
R	3	3	
W			2

	T1	T2
TS	3	2

Timestamp Example 1

	T1	T2
	Read(X)	Read(X)
	Read(Y)	Read(Y)
→	Y = Y + X	Z = Y - X
	Write(Y)	Write(Z)

	X	Y	Z
R	3	3	
W			2

	T1	T2
TS	3	2

Timestamp Example 1

	T1	T2
	Read(X)	Read(X)
	Read(Y)	Read(Y)
	Y = Y + X	Z = Y - X
→	Write(Y)	Write(Z)

	X	Y	Z
R	3	3	
W		3	2

	T1	T2
TS	3	2

Timestamp ordering – example 2

- Consider the following concurrent schedule

TS(T1) > (WTS(X) = ***0)****, read allowed;*
RTS(X) ← ***10***

TS(T1) > WTS(X) = ***0****;*
TS(T1) = RTS(X) = ***10****; write allowed;*
WTS(X) ← ***10***

1. Read(X)
2. X = X – k
3. Write(X)

TS(T2) > WTS(X) = ***10****, read allowed;*
RTS(X) ← ***20***

TS(T2) > WTS(X) = ***10****;*
TS(T2) = RTS(X) = ***20****; write allowed;*
WTS(X) ← ***20***

1. Read(X)
2. X = X * 1.01
3. Write(X)

TS(T1) > WTS(Y) = ***0****, read allowed;*
RTS(Y) ← ***10***

TS(T1) > WTS(Y) = ***0****;*
TS(T1) = RTS(Y) = ***10****; write allowed;*
WTS(Y) ← ***10***

4. Read(Y)
5. Y = Y + k
6. Write(Y)

4. Read(Y)
5. Y = Y * 1.01
6. Write(Y)

TS(T2) > WTS(Y) = ***10****, read allowed;*
RTS(Y) ← ***20***

TS(T2) > WTS(X) = ***10****;*
TS(T2) = RTS(Y) = ***20****; write allowed;*
WTS(Y) ← ***20***

RTS(X) :	**0**	**10**	**10**	**20**	**20**	**20**	**20**	**20**	**20**
WTS(X):	**0**	**0**	**10**	**10**	**20**	**20**	**20**	**20**	**20**
RTS(Y) :	**0**	**0**	**0**	**0**	**0**	**10**	**10**	**20**	**20**
WTS(Y):	**0**	**0**	**0**	**0**	**0**	**0**	**10**	**10**	**20**

T1 (TS = 10) **T2 (TS = 20)**

Thomas' write rule

- Write-write conflict may be acceptable in many cases
- Suppose T1 do a write(X) and then T2 do a write(X) and there is no transaction accessing X in between
- Then T2 only overwrite a value that is never being used
- In such case, it can be argued that such a write is acceptable

Thomas’ write rule

- In timestamp ordering, it is referred as the Thomas write rule:
- If a transaction T issue a write(X):
  - If TS(T) < RTS(X) then write is rejected, T has to **abort**
  - Else If TS(T) < WTS(X) then write is ***ignored***
  - Else, allow the write, and update WTS(X) accordingly

Timestamp

- The protocol means that transactions with higher times take precedence
  - Equivalent to running transactions in order of their final time values
  - Transactions don’t wait - no deadlock
- Problems
  - Long transactions might keep getting restarted by new transactions - starvation
  - Rolls back old transactions, which may have done a lot of work

Optimistic concurrency control

- 2PL & TSO are pessimistic protocols
    - They assume transactions will have problems
- Most optimistic point-of-view:
    - Assume no problem and let transaction execute
    - But before commit, do a final check
    - Only when a problem is discovered, then one aborts
- Basis for ***optimistic concurrency control***

Optimistic concurrency control

- Each transaction T is divided into 3 phases:
    1. **Read and execution**: T reads from the database and execute. However, T only writes to temporary location (not to the database itself)
    2. **Validation**: T checks whether there is conflict with other transaction, abort if necessary
    3. **Write** : T actually write the values in temporary location to the database
- Each transaction must follow the same order