CS 525: Advanced Database Organization

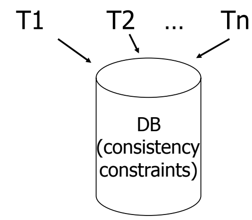
14: Concurrency Control

Boris Glavic

Slides: adapted from a course taught by

Hector Garcia-Molina, Stanford InfoLab

Chapter 18 [18] Concurrency Control



Example:

T1:	T2:
Read(A)	Read(A)
A ← A+100	A ← A×2
Write(A)	Write(A)
Read(B)	Read(B)
B ← B+100	B ← B×2
Write(B)	Write(B)

Constraint: A=B

Schedule A

T1	T2
Read(A); A ← A+100	
Write(A);	
Read(B); B ← B+100;	
Write(B);	
	Read(A);A ← A×2;
	Write(A);
	Read(B);B ← B×2;
	Write(B);

Schedule A

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedule B

T1	T2
	Read(A);A ← A×2;
	Write(A);
	Read(B);B ← B×2;
	Write(B);
Read(A); A ← A+100	
Write(A);	
Read(B); B ← B+100;	
Write(B);	

Schedule B

T1	T2	A	B
		25	25
	Read(A);A ← A×2;		
	Write(A);	50	
	Read(B);B ← B×2;		
	Write(B);		50
Read(A); A ← A+100			
Write(A);		150	
Read(B); B ← B+100;			
Write(B);			150
		150	150

Schedule C

T1	T2
Read(A); A ← A+100	
Write(A);	
	Read(A);A ← A×2;
	Write(A);
Read(B); B ← B+100;	
Write(B);	
	Read(B);B ← B×2;
	Write(B);

Schedule C

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
Read(B); B ← B+100;			
Write(B);			125
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedule D

T1	T2
Read(A); A ← A+100	
Write(A);	
	Read(A);A ← A×2;
	Write(A);
	Read(B);B ← B×2;
	Write(B);
Read(B); B ← B+100;	
Write(B);	

Schedule D

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		50
Read(B); B ← B+100;			
Write(B);			150
		250	150

Schedule E

Same as Schedule D but with new T2'

T1	T2'
Read(A); A ← A+100	
Write(A);	
	Read(A);A ← A×1;
	Write(A);
	Read(B);B ← B×1;
	Write(B);
Read(B); B ← B+100;	
Write(B);	

Schedule E

Same as Schedule D but with new T2'

T1	T2'	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×1;		
	Write(A);	125	
	Read(B);B ← B×1;		
	Write(B);		25
Read(B); B ← B+100;			
Write(B);			125
		125	125

Serial Schedules

- As long as we do not execute transactions in parallel and each transaction does not violate the constraints we are good
  - All schedules with no interleaving of transaction operations are called **serial** schedules

Definition: Serial Schedule

- No transactions are interleaved
  - There exists no two operations from transactions Ti and Tj so that both operations are executed before either transaction commits

$T_1 = r_1(A), w_1(A), r_1(B), w_1(B), c_1$

$T_2 = r_2(A), w_2(A), r_2(B), w_2(B), c_2$

Serial Schedule

$S_1 = r_2(A), w_2(A), r_2(B), w_2(B), c_2, r_1(A), w_1(A), r_1(B), w_1(B), c_1$

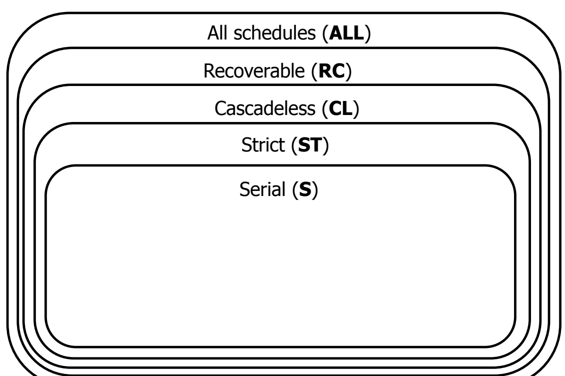
Nonserial Schedule

$S_2 = r_2(A), w_2(A), r_1(A), w_1(A), r_2(B), w_2(B), c_2, r_1(B), w_1(B), c_1$

Compare Classes

S ⊂ ST ⊂ CL ⊂ RC ⊂ ALL

- Abbreviations
  - S = Serial
  - ST = Strict
  - CL = Cascadeless
  - RC = Recoverable
  - ALL = all possible schedules



Why not serial schedules?

- No concurrency! ☹

- Want schedules that are "good", regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

Example:

$Sc=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

Outline

- Since serial schedules have good properties we would like our schedules to behave like (be **equivalent** to) serial schedules
  1. Need to define equivalence based solely on order of operations
  2. Need to define class of schedules which is equivalent to serial schedule
  3. Need to design scheduler that guarantees that we only get these good schedules

Example:

$Sc=r_1(A)w_1(A)\underbrace{r_2(A)w_2(A)}\underbrace{r_1(B)w_1(B)}r_2(B)w_2(B)$

$Sc'=\underbrace{r_1(A)w_1(A)\ r_1(B)w_1(B)}_{T_1}\underbrace{r_2(A)w_2(A)r_2(B)w_2(B)}_{T_2}$

However, for Sd:

$Sd=\underbrace{r_1(A)w_1(A)}r_2(A)w_2(A)\ r_2(B)w_2(B)\underbrace{r_1(B)w_1(B)}$

- as a matter of fact,

  T_2 must precede T_1

  in any equivalent schedule,

  i.e., $T_2 \rightarrow T_1$

- $T_2 \rightarrow T_1$
- Also, $T_1 \rightarrow T_2$

$T_1 \rightleftarrows T_2$ ⇨ Sd cannot be rearranged into a serial schedule

⇨ Sd is not "equivalent" to any serial schedule

⇨ Sd is "bad"

Returning to Sc

$Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

$T_1 \rightarrow T_2$ $\qquad$ $T_1 \rightarrow T_2$

Returning to Sc

$Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

$T_1 \rightarrow T_2$ $\qquad$ $T_1 \rightarrow T_2$

☛ no cycles ⇒ Sc is "equivalent" to a serial schedule (in this case T_1,T_2)

Concepts

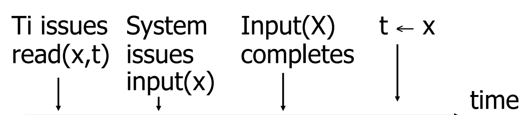
Transaction: sequence of $r_i(x)$, $w_i(x)$ actions

Conflicting actions: $r_1(A)$ / $w_2(A)$, $w_2(A)$ / $r_1(A)$, $w_1(A)$ / $w_2(A)$

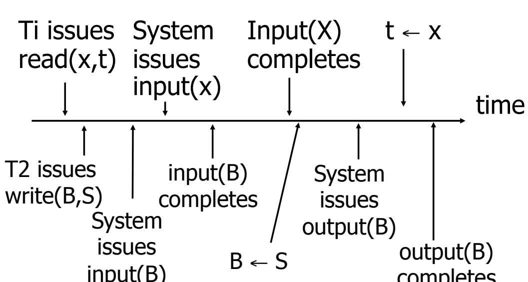
Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions

What about concurrent actions?



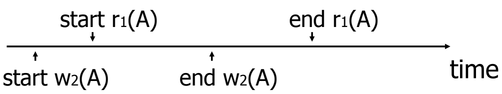
What about concurrent actions?



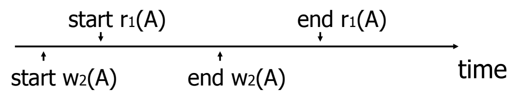
So net effect is either

- $S = \ldots r_1(x) \ldots w_2(b) \ldots$ or
- $S = \ldots w_2(B) \ldots r_1(x) \ldots$

What about conflicting, concurrent actions on same object?



What about conflicting, concurrent actions on same object?



- Assume equivalent to either $r_1(A)$ $w_2(A)$ or $w_2(A)$ $r_1(A)$
- $\Rightarrow$ low level synchronization mechanism
- Assumption called "atomic actions"

Outline

- Since serial schedules have good properties we would like our schedules to behave like (be **equivalent** to) serial schedules
  1. Need to define equivalence based solely on order of operations
  2. Need to define class of schedules which is equivalent to serial schedule
  3. Need to design scheduler that guarantees that we only get these good schedules

Conflict Equivalence

- Define equivalence based on the order of conflicting actions

Definition

S_1, S_2 are conflict equivalent schedules if S_1 can be transformed into S_2 by a series of swaps on non-conflicting actions.

Alternatively:

If the order of conflicting actions in S_1 and S_2 is the same

Outline

- Since serial schedules have good properties we would like our schedules to behave like (be **equivalent** to) serial schedules
  1. Need to define equivalence based solely on order of operations
  2. Need to define class of schedules which is equivalent to serial schedule
  3. Need to design scheduler that guarantees that we only get these good schedules

Definition

A schedule is conflict serializable (**CSR**) if it is conflict equivalent to some serial schedule.

How to check?

- Compare orders of all conflicting operations
- Can be simplified because there is some redundant information here, e.g.,

  $S_1 = w_2(A), w_2(B), r_1(A), w_1(B)$

  – W2(A) conflicts with R1(A)
  – W2(B) conflicts with W1(B)
  – Both imply that T2 has to be executed before T1 in any equivalent serial schedule

Conflict graph P(S) (S is schedule)

Nodes: transactions in S

Arcs: Ti → Tj whenever

- $p_i(A)$, $q_j(A)$ are actions in S
- $p_i(A) <_S q_j(A)$
- at least one of p_i, q_j is a write

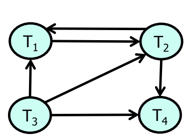
Exercise:

- What is P(S) for
  S = $w_3(A)$ $w_2(C)$ $r_1(A)$ $w_1(B)$ $r_1(C)$ $w_2(A)$ $r_4(A)$ $w_4(D)$

- Is S serializable?

Exercise:

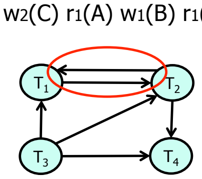
- What is P(S) for
  S = $w_3(A)$ $w_2(C)$ $r_1(A)$ $w_1(B)$ $r_1(C)$ $w_2(A)$ $r_4(A)$ $w_4(D)$



- Is S serializable?

Exercise:

- What is P(S) for
  S = $w_3(A)$ $w_2(C)$ $r_1(A)$ $w_1(B)$ $r_1(C)$ $w_2(A)$ $r_4(A)$ $w_4(D)$



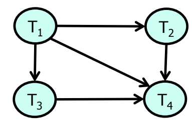
- Is S serializable?

Another Exercise:

- What is P(S) for
  $S = w_1(A)\ r_2(A)\ r_3(A)\ w_4(A)$?

Another Exercise:

- What is P(S) for
  $S = w_1(A)\ r_2(A)\ r_3(A)\ w_4(A)$?



Lemma

S_1, S_2 conflict equivalent $\Rightarrow P(S_1)=P(S_2)$

Lemma

S_1, S_2 conflict equivalent $\Rightarrow P(S_1)=P(S_2)$

Proof: ($a \rightarrow b$ same as $\neg b \rightarrow \neg a$)

Assume $P(S_1) \neq P(S_2)$

$\Rightarrow \exists\, T_i$: $T_i \rightarrow T_j$ in S_1 and not in S_2

$$\left.\begin{array}{l} \Rightarrow S_1 = \ldots p_i(A) \ldots q_j(A) \ldots \\ \quad\ S_2 = \ldots q_j(A) \ldots p_i(A) \ldots \end{array}\right\} \begin{array}{l} p_i,\ q_j \\ \text{conflict} \end{array}$$

$\Rightarrow S_1, S_2$ not conflict equivalent

Note: $P(S_1)=P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent

Note: $P(S_1)=P(S_2) \not\Rightarrow S_1, S_2$ conflict equivalent

Counter example:

$S_1=w_1(A)\ r_2(A) \quad w_2(B)\ r_1(B)$

$S_2=r_2(A)\ w_1(A) \quad r_1(B)\ w_2(B)$

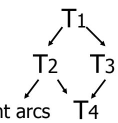
Theorem

$P(S_1)$ acyclic $\Longleftrightarrow$ S_1 conflict serializable

($\Leftarrow$) Assume S_1 is conflict serializable
$\Rightarrow \exists\, S_s$: S_s, S_1 conflict equivalent
$\Rightarrow P(S_s) = P(S_1)$
$\Rightarrow P(S_1)$ acyclic since $P(S_s)$ is acyclic

Theorem

$P(S_1)$ acyclic $\Longleftrightarrow$ S_1 conflict serializable



($\Rightarrow$) Assume $P(S_1)$ is acyclic
Transform S_1 as follows:
(1) Take T_1 to be transaction with no incident arcs
(2) Move all T_1 actions to the front

S_1 = ... $q_j(A)$.......$p_1(A)$.....

(3) we now have S_1 = < T_1 actions >< ... rest ...>
(4) repeat above steps to serialize rest!

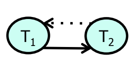
What's the damage?

- Classification of "bad" things that can happen in "bad" schedules
  - Dirty reads
  - Non-repeatable reads
  - Phantom reads (later)

Dirty Read

- A transaction T_1 read a value that has been updated by an uncommitted transaction T_2
- If T_2 aborts then the value read by T_1 is invalid

Dirty Read



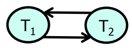
T_1	T_2	A=50
Read(A), A += 100		T_1: A = 150
Write(A);		A = 150
	Read(A), A +=200	T_2: A = 350
Abort		
	Write(A);	

$S_1 = r_1(A), w_1(A), r_2(A), a_1, w_2(A)$

Non-repeatable Read

- A transaction T_1 reads items; some before and some after an update of these item by a transaction T_2
- Problem
  - Repeated reads of the same item see different values
  - Some values are modified and some are not

Inconsistent Read



T_1	T_2	
		A = 100
Read(A)		
	Read(A), A /= 2	A = 50
	Write(A)	
	Commit	
Read(A)		A = 50
Commit		

$S_1 = r_1(A), r_2(A), w_2(A), c_2, r_1(A), c_1$

How to enforce serializable schedules?

Option 1: run system, recording P(S); at end of day, check for P(S) cycles and declare if execution was good

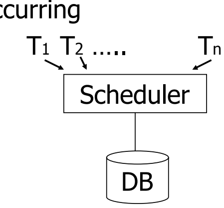
How to enforce serializable schedules?

Option 1: run system, recording P(S); at end of day, check for P(S) cycles and declare if execution was good

This is called **optimistic concurrency control**

How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring



How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring

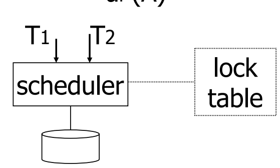
This is called **pessimistic concurrency control**

A locking protocol

Two new actions:

lock (exclusive): $l_i(A)$

unlock: $u_i(A)$



Rule #1: Well-formed transactions

T_i: … $l_i(A)$ … $p_i(A)$ … $u_i(A)$ …

1) Transaction has to lock A before it can access A
2) Transaction has to unlock A eventually
3) Transaction cannot access A after unlock

Rule #2 Legal scheduler

S = …….. $l_i(A)$ ………… $u_i(A)$ ………

no $l_j(A)$

4) Only one transaction can hold a lock on A at the same time

Exercise:

- What schedules are legal?
  What transactions are well-formed?

  $S_1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
  $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

  $S_2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$
  $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

  $S_3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
  $l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Exercise:

- What schedules are legal?
  What transactions are well-formed?

  $S1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$
  $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

  $S2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$
  $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$

  $S3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$
  $l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Schedule F

T1	T2
$l_1(A)$;Read(A)	
A←A+100;Write(A);$u_1(A)$	
	$l_2(A)$;Read(A)
	A←Ax2;Write(A);$u_2(A)$
	$l_2(B)$;Read(B)
	B←Bx2;Write(B);$u_2(B)$
$l_1(B)$;Read(B)	
B←B+100;Write(B);$u_1(B)$	

Schedule F

T1	T2	A	B
		25	25
$l_1(A)$;Read(A)			
A←A+100;Write(A);$u_1(A)$		125	
	$l_2(A)$;Read(A)		
	A←Ax2;Write(A);$u_2(A)$	250	
	$l_2(B)$;Read(B)		
	B←Bx2;Write(B);$u_2(B)$		50
$l_1(B)$;Read(B)			
B←B+100;Write(B);$u_1(B)$			150
		250	150

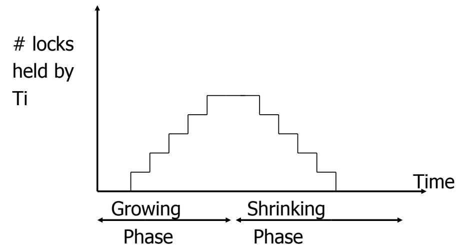
Rule #3 Two phase locking (**2PL**)

for transactions

T_i = ……. $l_i(A)$ ………... $u_i(A)$ ……...

no unlocks — no locks

5) A transaction does not require new locks after its first unlock operation



Schedule G

T1	T2
$l_1(A)$;Read(A)	
A←A+100;Write(A)	
$l_1(B)$; $u_1(A)$	
	$l_2(A)$;Read(A)
	A←Ax2;Write(A); $l_2(B)$ delayed

Schedule G

T1	T2
$l_1(A)$;Read(A)	
A←A+100;Write(A)	
$l_1(B)$; $u_1(A)$	
	$l_2(A)$;Read(A)
	A←Ax2;Write(A); $l_2(B)$ delayed
Read(B);B ← B+100	
Write(B); $u_1(B)$	

Schedule G

T1	T2
$l_1(A)$;Read(A)	
A←A+100;Write(A)	
$l_1(B)$; $u_1(A)$	
	$l_2(A)$;Read(A)
	A←Ax2;Write(A); $l_2(B)$ delayed
Read(B);B ← B+100	
Write(B); $u_1(B)$	
	$l_2(B)$; $u_2(A)$;Read(B)
	B ← Bx2;Write(B);$u_2(B)$;

Schedule H (T2 reversed)

T1	T2
$l_1(A)$; Read(A)	$l_2(B)$;Read(B)
A←A+100;Write(A)	B←Bx2;Write(B)
$l_1(B)$ delayed	$l_2(A)$ delayed

Deadlock

- Two or more transactions are waiting for each other to release a lock
- In the example
  - T_1 is waiting for T_2 and is making no progress
  - T_2 is waiting for T_1 and is making no progress
  - -> if we do not do anything they would wait forever

- Assume deadlocked transactions are rolled back
  - They have no effect
  - They do not appear in schedule
  - **Come back to that later**

E.g., Schedule H = $\underbrace{\qquad\qquad\qquad}_{\text{This space intentionally left blank!}}$

<u>Next step:</u>

Show that rules #1,2,3 $\Rightarrow$ conflict-serializable schedules

<u>Conflict rules for</u> $l_i(A)$, $u_i(A)$:

- $l_i(A)$, $l_j(A)$ conflict
- $l_i(A)$, $u_j(A)$ conflict

Note: no conflict < $u_i(A)$, $u_j(A)$>, < $l_i(A)$, $r_j(A)$>,...

<u>Theorem</u> Rules #1,2,3 (2PL) $\Rightarrow$ conflict serializable schedule

<u>Theorem</u> Rules #1,2,3 (2PL) $\Rightarrow$ conflict serializable schedule

To help in proof:

<u>Definition</u> Shrink(Ti) = SH(Ti) = first unlock action of Ti

Lemma

Ti → Tj in S ⇒ SH(Ti) $<_S$ SH(Tj)

Lemma

Ti → Tj in S ⇒ SH(Ti) $<_S$ SH(Tj)

Proof of lemma:

Ti → Tj means that

S = … $p_i(A)$ … $q_j(A)$ …; p,q conflict

By rules 1,2:

S = … $p_i(A)$ … $u_i(A)$ … $l_j(A)$ **…** $q_j(A)$ …

Lemma

Ti → Tj in S ⇒ SH(Ti) $<_S$ SH(Tj)

Proof of lemma:

Ti → Tj means that

S = … $p_i(A)$ … $q_j(A)$ …; p,q conflict

By rules 1,2:

S = … $p_i(A)$ … $u_i(A)$ … $l_j(A)$ **…** $q_j(A)$ …

By rule 3: SH(Ti) SH(Tj)

So, SH(Ti) $<_S$ SH(Tj)

Theorem Rules #1,2,3 ⇒ conflict serializable schedule (2PL)

Proof:

(1) Assume P(S) has cycle

$T_1 \to T_2 \to \dots T_n \to T_1$

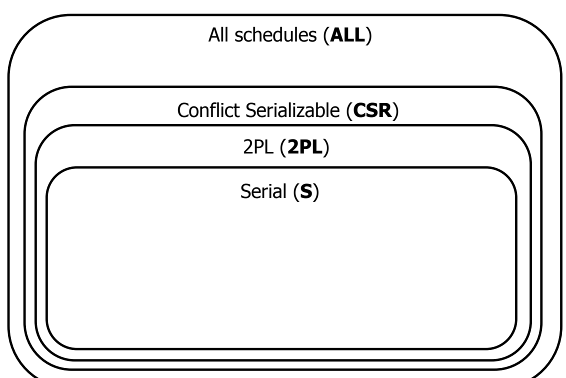
(2) By lemma: $SH(T_1) < SH(T_2) < \dots < SH(T_1)$

(3) Impossible, so P(S) acyclic

(4) ⇒ S is conflict serializable

2PL subset of Serializable

S ⊂ 2PL⊂ CSR⊂ ALL



S1: w1(x) w3(x) w2(y) w1(y)

- S1 cannot be achieved via 2PL: The lock by T1 for y must occur after w2(y), so the unlock by T1 for x must occur after this point (and before w1(x)). Thus, w3(x) cannot occur under 2PL where shown in S1 because T1 holds the x lock at that point.
- However, S1 is serializable (equivalent to T2, T1, T3).

If you need a bit more practice:

Are our schedules S_C and S_D 2PL schedules?

S_C: w1(A) w2(A) w1(B) w2(B)

S_D: w1(A) w2(A) w2(B) w1(B)

- Beyond this simple **2PL** protocol, it is all a matter of improving performance and allowing more concurrency….
  - Shared locks
  - Multiple granularity
  - Avoid Deadlocks
  - Inserts, deletes and phantoms
  - Other types of C.C. mechanisms
    - Multiversioning concurrency control

Shared locks

So far:

S = ...l_1(A) r_1(A) u_1(A) … l_2(A) r_2(A) u_2(A) …

Do not conflict

Shared locks

So far:

S = ...l_1(A) r_1(A) u_1(A) … l_2(A) r_2(A) u_2(A) …

Do not conflict

Instead:

S=... ls_1(A) r_1(A) ls_2(A) r_2(A) …. us_1(A) us_2(A)

Lock actions

l-t_i(A): lock A in t mode (t is S or X)

u-t_i(A): unlock t mode (t is S or X)

Shorthand:

u_i(A): unlock whatever modes T_i has locked A

Rule #1 Well formed transactions

T_i =... l-S_1(A) … r_1(A) … u_1 (A) …
T_i =... l-X_1(A) … w_1(A) … u_1 (A) …

- What about transactions that read and write same object?

Option 1: Request exclusive lock
T_i = ...l-X_1(A) … r_1(A) ... w_1(A) ... u(A) …

- What about transactions that read and write same object?

Option 2: Upgrade

(E.g., need to read, but don't know if will write…)

T_i=... l-S_1(A) … r_1(A) ... l-X_1(A) …w_1(A) ...u(A)…

Think of
- Get 2nd lock on A, or
- Drop S, get X lock

Rule #2 Legal scheduler

S = ...l-S_i(A) … … u_i(A) …

no l-X_j(A)

S = ... l-X_i(A) … … u_i(A) …

no l-X_j(A)
no l-S_j(A)

A way to summarize Rule #2

Compatibility matrix

Comp

	S	X
S	true	false
X	false	false

Rule # 3 2PL transactions

No change except for upgrades:
(I) If upgrade gets more locks
(e.g., S → {S, X}) then no change!
(II) If upgrade releases read (shared) lock (e.g., S → X)
- can be allowed in growing phase

<u>Theorem</u> Rules 1,2,3 ⇒ Conf.serializable
for S/X locks schedules

<u>Proof:</u> similar to X locks case

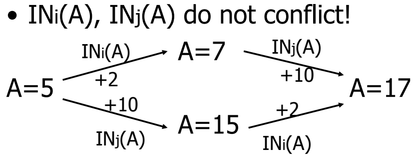
<u>Detail:</u>
l-t_i(A), l-r_j(A) do not conflict if comp(t,r)
l-t_i(A), u-r_j(A) do not conflict if comp(t,r)

<u>Lock types beyond S/X</u>

Examples:
(1) increment lock
(2) update lock

Example (1): increment lock

- Atomic increment action: IN_i(A)
  {Read(A); A ← A+k; Write(A)}
- IN_i(A), IN_j(A) do not conflict!



Comp

	S	X	I
S			
X			
I			

Comp

	S	X	I
S	T	F	F
X	F	F	F
I	F	F	T

<u>Update locks</u>

A common deadlock problem with upgrades:

T1	T2
l-S_1(A)	
	l-S_2(A)
l-X_1(A)	
	l-X_2(A)

--- Deadlock ---

Solution

If T_i wants to read A and knows it may later want to write A, it requests update lock (not shared)

Comp

New request

Lock already held in	S	X	U
S			
X			
U			

Comp

New request

Lock already held in	S	X	U
S	T	F	T
X	F	F	F
U	TorF	F	F

-> symmetric table?

Note: object A may be locked in different modes at the same time...

S_1=...l-S_1(A)...l-S_2(A)...l-U_3(A)... { l-S_4(A)...?
l-U_4(A)...?

Note: object A may be locked in different modes at the same time...

S_1=...l-S_1(A)...l-S_2(A)...l-U_3(A)... { l-S_4(A)...?
l-U_4(A)...?

- To grant a lock in mode t, mode t must be compatible with all currently held locks on object

How does locking work in practice?

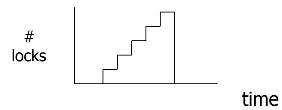
- Every system is different

  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way ...

Sample Locking System:

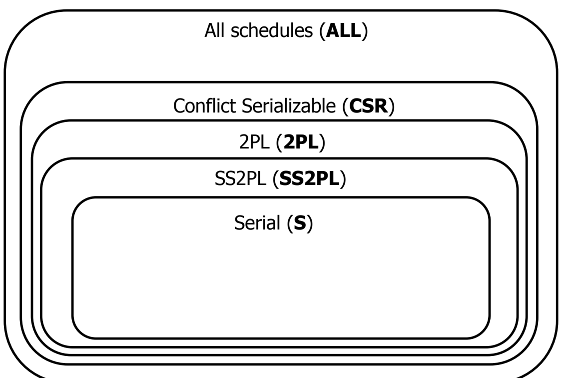
(1) Don't trust transactions to request/release locks

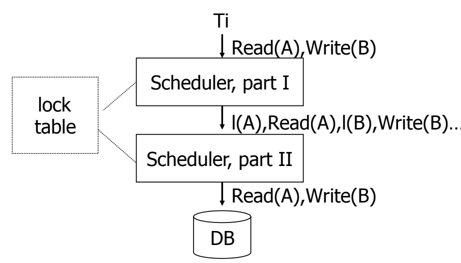
(2) Hold all locks until transaction commits



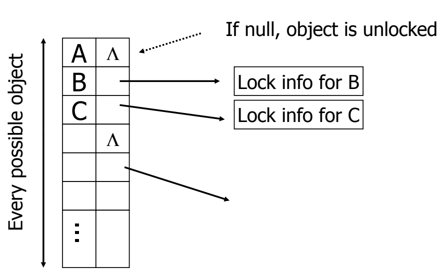
Strict Strong 2PL (**SS2PL**)

- 2PL + (2) from the last slide
- All locks are held until transaction end
- Compare with schedule class **strict** (**ST**) we defined for recovery
  - A transaction never reads or writes items written by an uncommitted transactions
- **SS2PL** = (**ST** ∩ **2PL**)

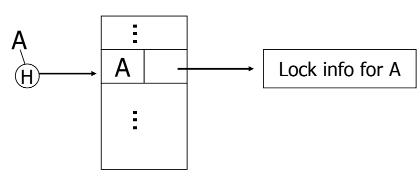




Lock table Conceptually

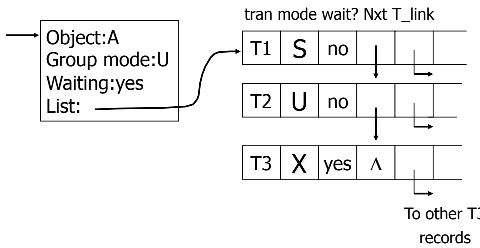


But use hash table:

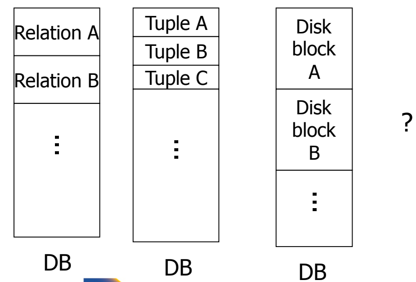


If object not found in hash table, it is unlocked

Lock info for A - example



What are the objects we lock?

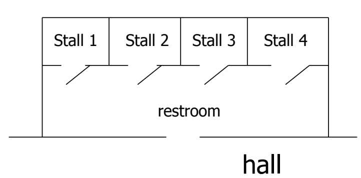


- Locking works in any case, but should we choose small or large objects?

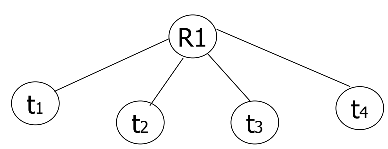
- Locking works in any case, but should we choose small or large objects?
- If we lock large objects (e.g., Relations)
  - Need few locks
  - Low concurrency
- If we lock small objects (e.g., tuples,fields)
  - Need more locks
  - More concurrency

We can have it both ways!!

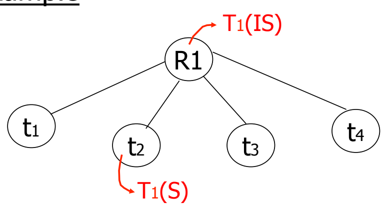
Ask any janitor to give you the solution...



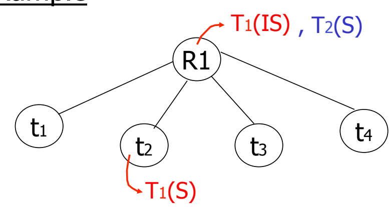
Example



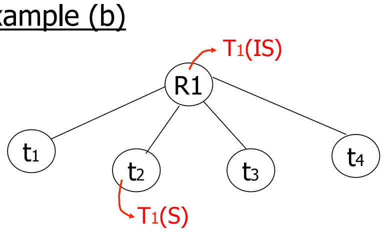
Example



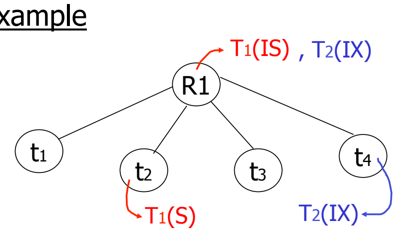
Example



Example (b)



Example



Multiple granularity

Comp — Requestor

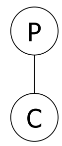
Holder	IS	IX	S	SIX	X
IS					
IX					
S					
SIX					
X					

Multiple granularity

Comp — Requestor

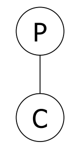
Holder	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

Parent locked in	Child can be locked in
IS	
IX	
S	
SIX	
X	



Parent locked in	Child can be locked by same transaction in
IS	IS, S
IX	IS, S, IX, X, SIX
S	none
SIX	X, IX, [SIX]
X	none

not necessary

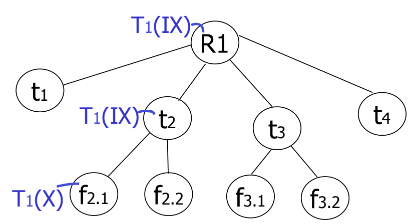


Rules

(1) Follow multiple granularity comp function
(2) Lock root of tree first, any mode
(3) Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
(4) Node Q can be locked by Ti in X,SIX,IX only if parent(Q) locked by Ti in IX,SIX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q's children are locked by Ti

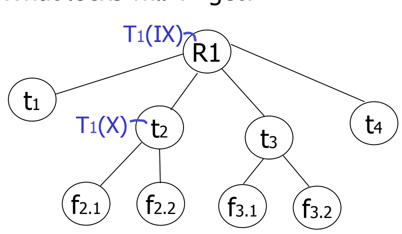
Exercise:

- Can T_2 access object $f_{2.2}$ in X mode? What locks will T_2 get?



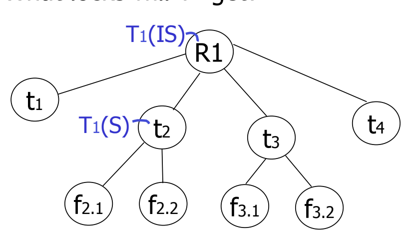
Exercise:

- Can T_2 access object $f_{2.2}$ in X mode? What locks will T_2 get?



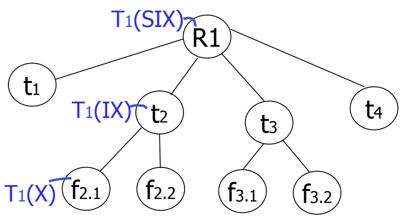
Exercise:

- Can T_2 access object $f_{3.1}$ in X mode? What locks will T_2 get?



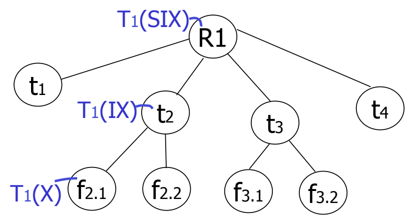
Exercise:

- Can T_2 access object $f_{2.2}$ in S mode? What locks will T_2 get?

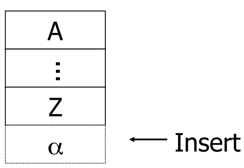


Exercise:

- Can T_2 access object $f_{2.2}$ in X mode? What locks will T_2 get?



Insert + delete operations



Modifications to locking rules:

(1) Get exclusive lock on A before deleting A

(2) At insert A operation by Ti, Ti is given exclusive lock on A

Still have a problem: **Phantoms**

Example: relation R (E#,name,…)
constraint: E# is key
use tuple locking

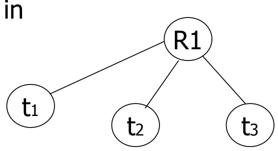
R	E#	Name	….
o1	55	Smith	
o2	75	Jones	

T_1: Insert <08,Obama,…> into R
T_2: Insert <08,McCain,…> into R

T_1	T_2
$S_1(o_1)$	$S_2(o_1)$
$S_1(o_2)$	$S_2(o_2)$
Check Constraint	Check Constraint
⋮	⋮
Insert o_3[08,Obama,..]	
	Insert o_4[08,McCain,..]

Solution

- Use multiple granularity tree
- Before insert of node Q, lock parent(Q) in X mode



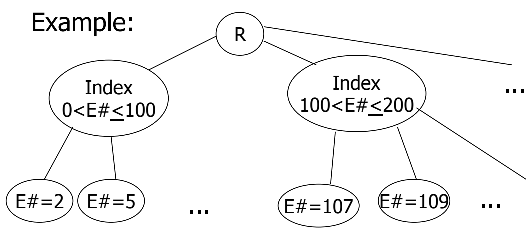
Back to example

T1: Insert<04,Kerry> T2: Insert<04,Bush>

T_1	T_2
$X_1(R)$	
	$X_2(R)$ *delayed*
Check constraint Insert<04,Kerry> U(R)	
	$X_2(R)$ Check constraint Oops! e# = 04 already in R!

Instead of using R, can use index on R:

Example:



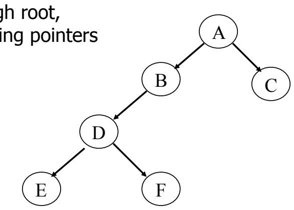
- This approach can be generalized to multiple indexes...

Next:

- Tree-based concurrency control
- Validation concurrency control

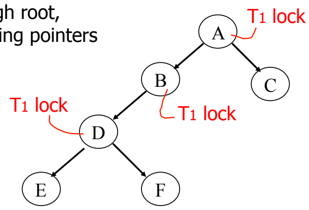
Example

- all objects accessed through root, following pointers



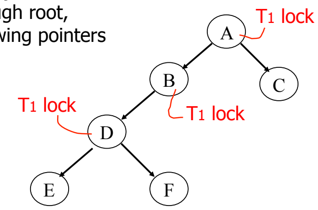
Example

- all objects accessed through root, following pointers



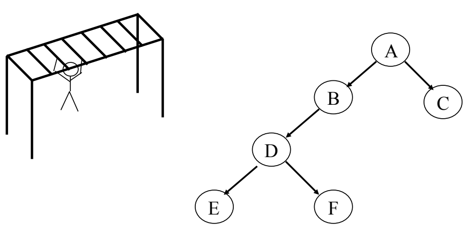
Example

- all objects accessed through root, following pointers

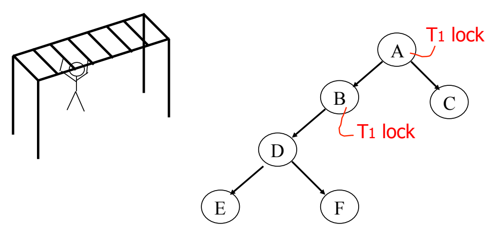


☛ can we release A lock if we no longer need A??

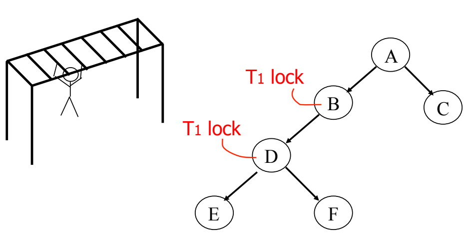
Idea: traverse like "Monkey Bars"



Idea: traverse like "Monkey Bars"

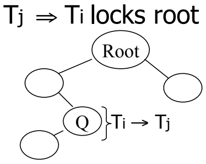


Idea: traverse like "Monkey Bars"



Why does this work?

- Assume all T_i start at root; exclusive lock
- $T_i \rightarrow T_j \Rightarrow T_i$ locks root before T_j



- Actually works if we don't always start at root

Rules: tree protocol (exclusive locks)

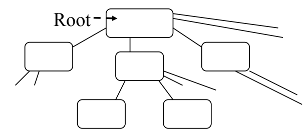
(1) First lock by T_i may be on any item

(2) After that, item Q can be locked by T_i only if parent(Q) locked by T_i

(3) Items may be unlocked at any time

(4) After T_i unlocks Q, it cannot relock Q

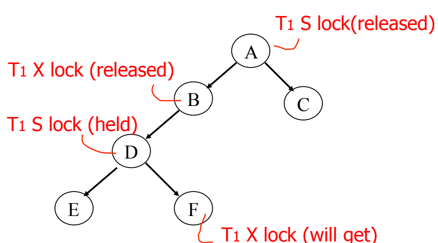
- Tree-like protocols are used typically for B-tree concurrency control



E.g., during insert, do not release parent lock, until you are certain child does not have to split

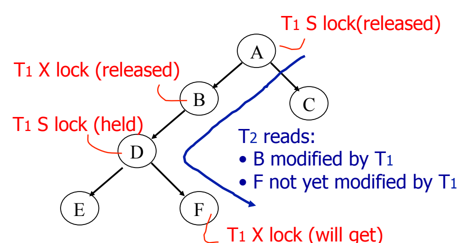
Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



Tree Protocol with Shared Locks

- Rules for shared & exclusive locks?



Tree Protocol with Shared Locks

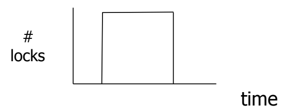
- Need more restrictive protocol
- Will this work??
  - Once T_1 locks one object in X mode, all further locks down the tree must be in X mode

Deadlocks (again)

- Before we assumed that we are able to detect deadlocks and resolve them
- Now two options
  - (1) Deadlock detection (and resolving)
  - (2) Deadlock prevention

Deadlock Prevention

- Option 1:
  - 2PL + transaction has to acquire all locks at transaction start following a global order



Deadlock Prevention

- Option 1:
  - Long lock durations ☹
  - Transaction has to know upfront what data items it will access ☹
    - E.g.,

```
UPDATE R SET a = a + 1 WHERE b < 15
```

    - We don't know what tuples are in R!

Deadlock Prevention

- Option 2:
  - Define some global order of data items O
  - Transactions have to acquire locks according to this order
- Example (X < Y < Z)

  $l_1(X)$, $l_1(Z)$ (OK)

  $l_1(Y)$, $l_1(X)$ (NOT OK)

Deadlock Prevention

- Option 2:
  - Accessed data items have to be known upfront ☹
  - or access to data has to follow the order ☹

Deadlock Prevention

- Option 3 (**Preemption**)
  - Roll-back transactions that wait for locks under certain conditions
  - 3 a) **wait-die**
    - Assign timestamp to each transaction
    - If transaction T_i waits for T_j to release a lock
      - Timestamp $T_i < T_j$ -> wait
      - Timestamp $T_i > T_j$ -> roll-back T_i

Deadlock Prevention

- Option 3 (**Preemption**)
  - Roll-back transactions that wait for locks under certain conditions
  - 3 a) **wound-wait**
    - Assign timestamp to each transaction
    - If transaction T_i waits for T_j to release a lock
      - Timestamp $T_i < T_j$ -> roll-back T_j
      - Timestamp $T_i > T_j$ -> wait

Deadlock Prevention

- Option 3:
  - Additional transaction roll-backs ☹

Timeout-based Scheme

- Option 4:
  - After waiting for a lock longer than X, a transaction is rolled back

Timeout-based Scheme

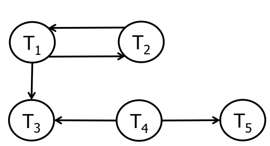
- Option 4:
  - Simple scheme ☺
  - Hard to find a good value of X
    - To high: long wait times for a transaction before it gets eventually aborted
    - To low: to many transaction that are not deadlock get aborted

Deadlock Detection and Resolution

- Data structure to detect deadlocks: **wait-for** graph
  - One node for each transaction
  - Edge T_i->T_j if T_i is waiting for T_j
  - Cycle -> Deadlock
    - Abort one of the transaction in cycle to resolve deadlock

Deadlock Detection and Resolution

- When do we run the detection?
- How to choose the victim?



<u>Optimistic Concurrency Control: Validation</u>

Transactions have 3 phases:

(1) <u>Read</u>
  - all DB values read
  - writes to temporary storage
  - no locking

(2) <u>Validate</u>
  - check if schedule so far is serializable

(3) <u>Write</u>
  - if validate ok, write to DB

Key idea

- Make validation atomic
- If $T_1, T_2, T_3, \ldots$ is validation order, then resulting schedule will be conflict equivalent to $S_s = T_1\ T_2\ T_3 \ldots$

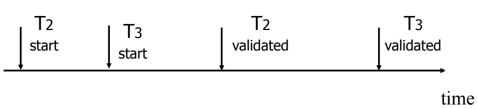
To implement validation, system keeps <u>two sets:</u>

- <u>FIN</u> = transactions that have finished phase 3 (and are all done)
- <u>VAL</u> = transactions that have successfully finished phase 2 (validation)

Example of what validation must prevent:

$RS(T_2)=\{B\}$ $\quad$ $RS(T_3)=\{A,B\}$ $\neq \phi$

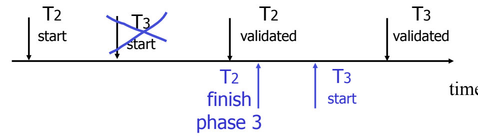
$WS(T_2)=\{B,D\}$ $\cap$ $WS(T_3)=\{C\}$



Example of what validation must ~~prevent~~ allow:

$RS(T_2)=\{B\}$ $\quad$ $RS(T_3)=\{A,B\}$ $\neq \phi$

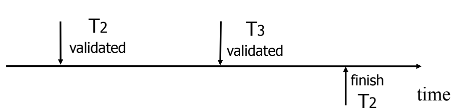
$WS(T_2)=\{B,D\}$ $\cap$ $WS(T_3)=\{C\}$



Another thing validation must prevent:

$RS(T_2)=\{A\}$ $\quad$ $RS(T_3)=\{A,B\}$

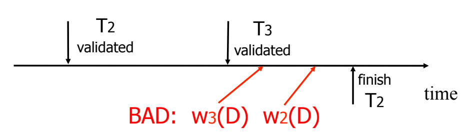
$WS(T_2)=\{D,E\}$ $\quad$ $WS(T_3)=\{C,D\}$



Another thing validation must prevent:

$RS(T_2)=\{A\}$ $\quad$ $RS(T_3)=\{A,B\}$

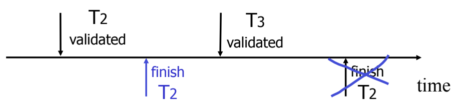
$WS(T_2)=\{D,E\}$ $\quad$ $WS(T_3)=\{C,D\}$



Another thing validation must ~~prevent~~ allow:

$RS(T_2)=\{A\}$ $RS(T_3)=\{A,B\}$

$WS(T_2)=\{D,E\}$ $WS(T_3)=\{C,D\}$



Validation rules for Tj:

(1) When T_j starts phase 1:

ignore(T_j) $\leftarrow$ FIN

(2) at T_j Validation:

if check (T_j) then

[VAL $\leftarrow$ VAL U $\{T_j\}$;

do write phase;

FIN $\leftarrow$FIN U $\{T_j\}$]

Check (T_j):

For $T_i \in$ VAL - IGNORE (T_j) DO

IF [$WS(T_i) \cap RS(T_j) \neq \emptyset$ OR

$T_i \notin$ FIN] THEN RETURN false;

RETURN true;

Check (T_j):

For $T_i \in$ VAL - IGNORE (T_j) DO

IF [$WS(T_i) \cap RS(T_j) \neq \emptyset$ OR

$T_i \notin$ FIN] THEN RETURN false;

RETURN true;

Is this check too restrictive ?

Improving Check(T_j)

For $T_i \in$ VAL - IGNORE (T_j) DO

IF [$WS(T_i) \cap RS(T_j) \neq \emptyset$ OR

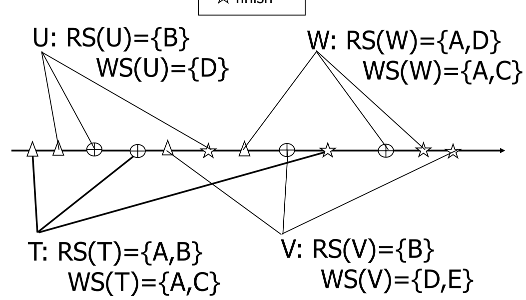
($T_i \notin$ FIN AND $WS(T_i) \cap WS(T_j) \neq \emptyset$)]

THEN RETURN false;

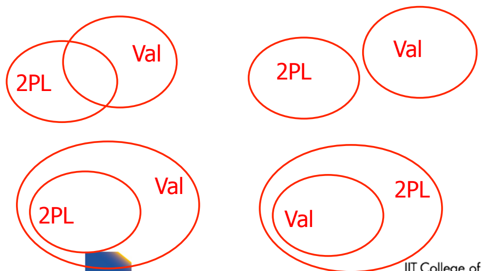
RETURN true;

Exercise:

$\triangle$ start
$\oplus$ validate
$\star$ finish



Is Validation = 2PL?



S2: w2(y) w1(x) w2(x)

- S2 can be achieved with 2PL:
  l2(y) w2(y) l1(x) w1(x) u1(x) l2(x) w2(x) u2(y) u2(x)
- S2 cannot be achieved by validation:
  The validation point of T2, val2 must occur before w2(y) since transactions do not write to the database until after validation. Because of the conflict on x, val1 < val2, so we must have something like
  S2: val1 val2 w2(y) w1(x) w2(x)
  With the validation protocol, the writes of T2 should not start until T1 is all done with its writes, which is not the case.

Validation subset of 2PL?

- Possible proof (Check!):
  - Let S be validation schedule
  - For each T in S insert lock/unlocks, get S':
    - At T start: request read locks for all of RS(T)
    - At T validation: request write locks for WS(T); release read locks for read-only objects
    - At T end: release all write locks
  - Clearly transactions well-formed and 2PL
  - Must show S' is legal (next page)

- Say S' not legal:
  S': ... l1(x) w2(x) r1(x) val1 u2(x) ...
  - At val1: T2 not in Ignore(T1); T2 in VAL
  - T1 does not validate: WS(T2) $\cap$ RS(T1) $\neq \emptyset$
  - contradiction!
- Say S' not legal:
  S': ... val1 l1(x) w2(x) w1(x) u2(x) ...
  - Say T2 validates first (proof similar in other case)
  - At val1: T2 not in Ignore(T1); T2 in VAL
  - T1 does not validate:
    T2 $\notin$ FIN AND WS(T1) $\cap$ WS(T2) $\neq \emptyset$)
  - contradiction!

Validation (also called **optimistic concurrency control**) is useful in some cases:

- Conflicts rare
- System resources plentiful
- Have real time constraints

Multiversioning Concurrency Control (MVCC)

- Keep old versions of data item and use this to increase concurrency
- Each write creates a new version of the written data item
- Use version numbers of timestamps to identify versions

Multiversioning Concurrency Control (MVCC)

- **Different transactions** operate over **different versions** of data items
- -> readers never have to wait for writers
- -> great for combined workloads
  - **OLTP** workload (writes, only access small number of tuples, short)
  - **OLAP** workload (reads, access large portions of database, long running)

MVCC schemes

- MVCC timestamp ordering
- MVCC 2PL
- Snapshot isolation (SI)
  - We will only cover this one

Snapshot Isolation (SI)

- Each transaction **T** is assigned a timestamp **S(T)** when it starts
- Each write creates a new data item version timestamped with the current timestamp
- When a transaction commits, then the latest versions created by the transaction get a timestamp **C(T)** as of the commit

Snapshot Isolation (SI)

- Under snapshot isolation each transaction T sees a consistent snapshot of the database as of S(T)
  - It only sees data item versions of transactions that committed before T started
  - It also sees its own changes

First Updater Wins Rule (FUW)

- Two transactions Ti and Tj may update the same data item A
  - To avoid lost updates only one of the two can be safely committed
- **First Updater Wins Rules**
  - The transaction that updated A first is allowed to commit
  - The other transaction is aborted

First Committer Wins Rule (FCW)

- Two transactions Ti and Tj may update the same data item A
  - To avoid lost updates only one of the two can be safely committed
- **First Committer Wins Rules**
  - The transaction that attempts to commit first is allowed to commit
  - The other transaction is aborted

©Silberschatz, Korth and Sudarshan

	X	Y	Z		T1	T2	T3
Update not visible outside of T1	0	1	5	1	W(Y := 1)		
Update becomes visible to future transactions	0			2	Commit		
				3		Start	
				4		R(X) → 0	
				5		R(Y)→ 1	
				6			
	2		3	7			W(X:=2)
	2		3	8			W(Z:=3)
				9			Commit
Concurrent updates not visible				10		R(Z) → 5	
				11		R(Y) → 1	
Not first-committer of X	3			12		W(X:=3)	
				13		Commit-Req	
Serialization error, T2 is rolled back				14		Abort	

Why does that work?

- Since all transactions see a consistent snapshot and their changes are only made "public" once they commit
  - It looks like the transactions have been executed in the order of their commits*

* Recall the writes to the same data item are disallowed for concurrent transactions

Is that serializable?

- Almost ;-)
- There is still one type of conflict which cannot occur in serialize schedules called **write-skew**

Write Skew

- Consider two data items A and B
  - A = 5, B = 5
- Concurrent Transactions T1 and T2
  - T1: A = A + B
  - T2: B = A + B
- Final result under SI
  - A = 10, B = 10

Write Skew

- Consider serial schedules:
  - T1, T2: A=10, B=15
  - T2, T1: A=15, B=10
- What is the problem
  - Under SI both T1 and T2 do not see each others changes
  - In any serial schedule one of the two would see the others changes

Example: Oracle

- Tuples are updated in place
- Old versions in separate ROLLBACK segment
  - GC once nobody needs them anymore
- How to implement the FCW or FUW?
  - Oracle uses write locks to block concurrent writes
  - Transaction waiting for a write lock aborts if transaction holding the lock commits

SI Discussion

- Advantages
  - Readers and writers do not block each other
  - If we do not GC old row versions we can go back to previous versions of the database -> Time travel
    - E.g., show me the customer table as it was yesterday
- Disadvantages
  - Storage overhead to keep old row versions
  - GC overhead
  - Not strictly serializable

Summary

Have studied CC mechanisms used in practice

- 2 PL variants
- Multiple lock granularity
- Deadlocks
- Tree (index) protocols
- Optimistic CC (Validation)
- Multiversioning Concurrency Control (MVCC)