CSCD43: Database Systems Technology

Lecture 13

Wael Aboulsaadat

Acknowledgment: these slides are based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the course's textbook.

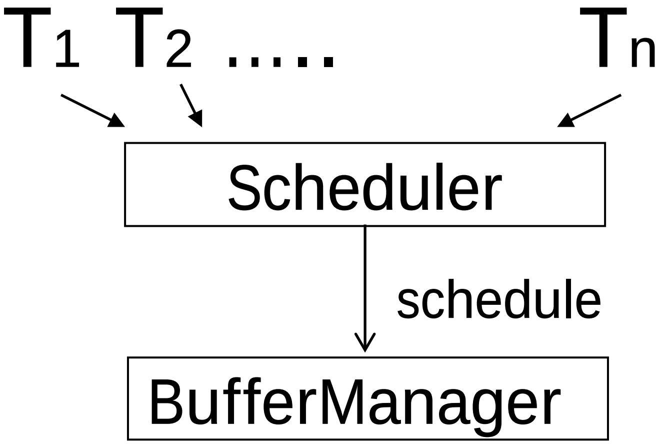
Enforcing Serializability

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! Unrealistic...!

How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring



But how ???

2.A) Buffer transactions during n seconds, stop DBMS, make schedule, execute schedule, repeat...

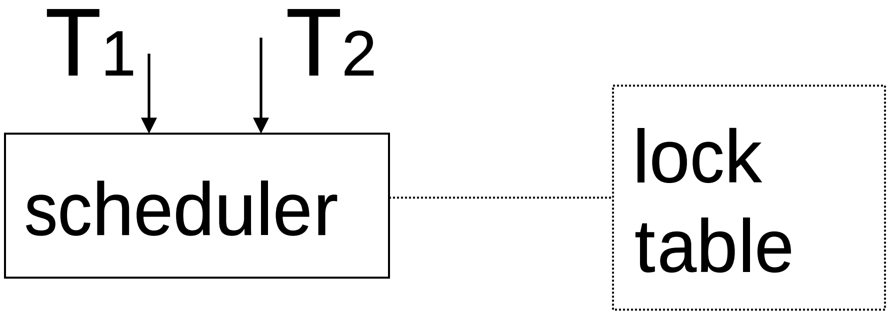
unrealistic...!

2.B) Use a locking protocol!

Two new actions:

lock (exclusive): l_i (A)

unlock: u_i (A)



T_1 T_2

scheduler

lock table

Rule #1: Well-formed transactions

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

Rule #2 Legal scheduler

$$S = \text{........}\ l_i(A)\ \underset{\text{no } l_j(A)}{\underleftrightarrow{\text{............}}}\ u_i(A)\ \text{.........}$$

Locking Example

T1: Read(A); A =A+100; Write(A);
Read(B); B=B+100; Write(B);

T2: Read(A); A =A*2; Write(A);
Read(B); B=B*2; Write(B);

Serial Schedule

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←Ax2;Write(A);	250	
	Read(B)		
	B←Bx2;Write(B);		250
		250	250

Schedule A

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 A

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

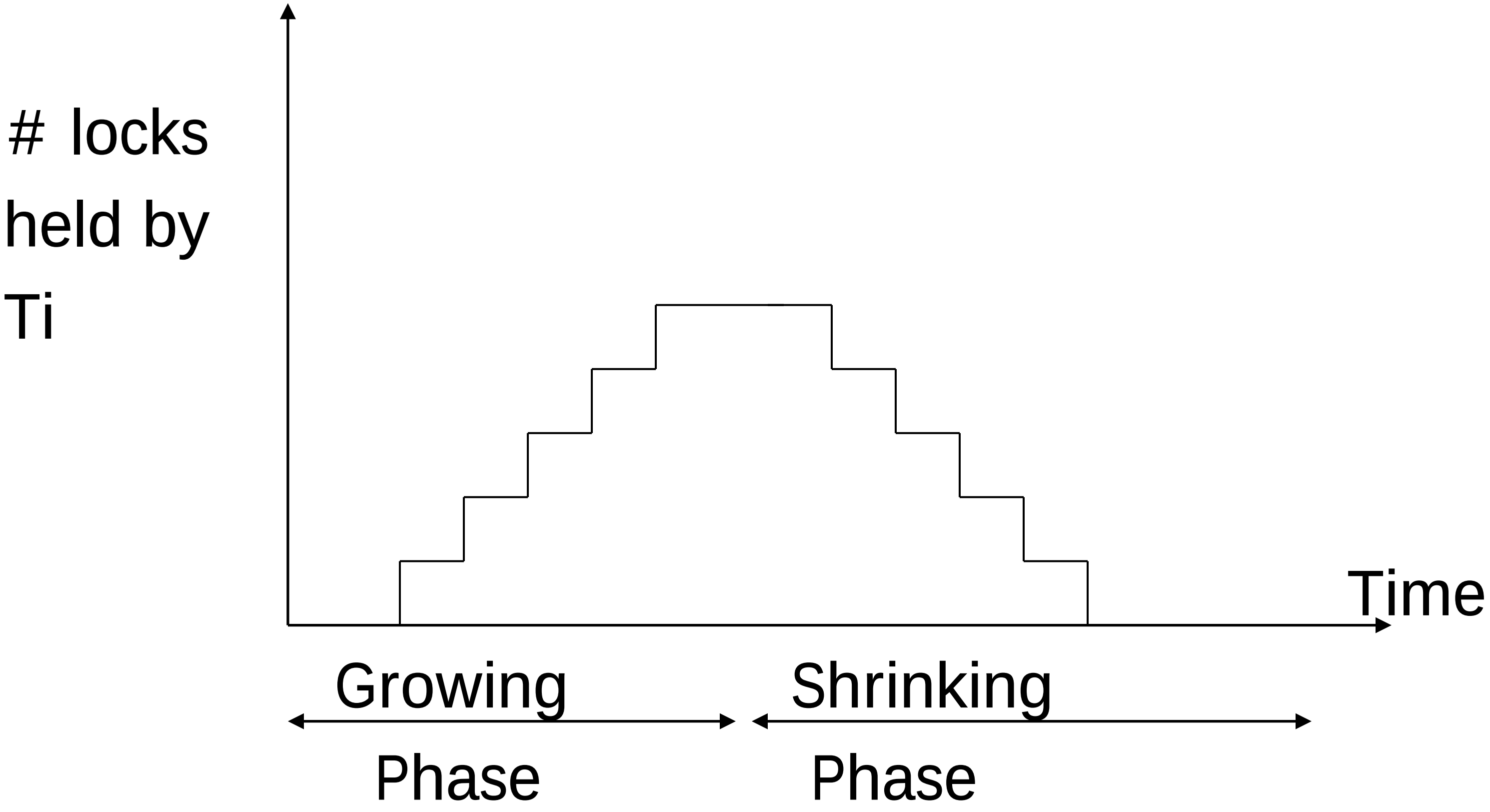
Rule #3 Two phase locking (2PL)

for transactions

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

no unlocks

no locks


locks
held by
Ti
Time
Growing
Phase
Shrinking
Phase

Schedule B

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

delayed

Schedule B

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 B

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

Schedule C (T$_2$ reversed)

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

- Assume deadlocked transactions are rolled back
  - They have no effect
  - They do not appear in schedule

Next step:

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

Conflict rules for $l_i(A)$, $u_i(A)$:

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

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

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

To help in proof:

Definition Shrink(Ti) = SH(Ti) =
first unlock action of Ti

Lemma

$T_i \rightarrow T_j$ in $S \Rightarrow SH(T_i) <_S SH(T_j)$

Proof of lemma:

$T_i \rightarrow T_j$ means that

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

By rules 1,2:

$S = \ldots\ p_i(A)\ \ldots\ u_i(A)\ \ldots\ l_j(A)\ \ldots\ q_j(A)\ \ldots$

By rule 3: $SH(T_i)$ $SH(T_j)$

So, $SH(T_i) <_S SH(T_j)$

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

Proof:

(1) Assume P(S) has cycle

$$T_1 \rightarrow T_2 \rightarrow .... T_n \rightarrow T_1$$

(2) By lemma: $SH(T_1) < SH(T_2) < ... < SH(T_1)$

(3) Impossible, so P(S) acyclic

(4) ⇒ S is conflict serializable

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

Shared locks

So far:

$S = \ldots l_1(A)\ r_1(A)\ u_1(A)\ \ldots\ l_2(A)\ r_2(A)\ u_2(A)\ \ldots$

Do not conflict

Instead:

$S = \ldots\ ls_1(A)\ r_1(A)\ ls_2(A)\ r_2(A)\ \ldots.us_1(A)us_2(A)$

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

- 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) ...

<u>Rule #1</u> 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 2: Upgrade

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

$T_i = \ldots\ l\text{-}S_1(A)\ \ldots\ r_1(A)\ \ldots\ l\text{-}X_1(A)\ \ldots w_1(A)\ \ldots u(A) \ldots$

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

Rule #2 Legal scheduler

$$S = \ldots.l\text{-}S_i(A)\ \underset{\longleftrightarrow}{\ldots\quad\ldots}\ u_i(A)\ \ldots$$

no l-$X_j(A)$

$$S = \ldots\ l\text{-}X_i(A)\ \underset{\longleftrightarrow}{\ldots\qquad\ldots}\ u_i(A)\ \ldots$$

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

A way to summarize Rule #2

Compatibility matrix

New request

Lock already held in

	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 $\Rightarrow$ 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)

Lock types beyond S/X

Examples:

(1) increment lock

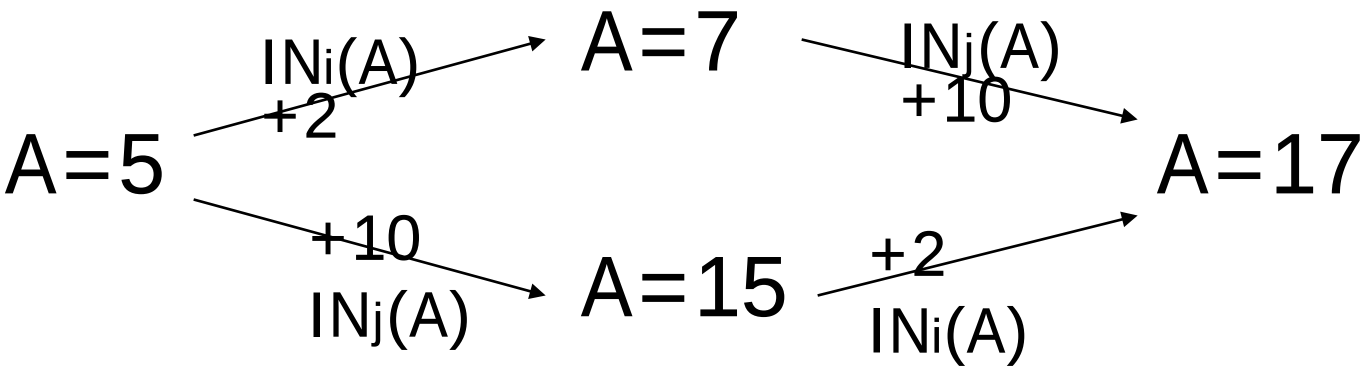
(2) update lock

Example (1): increment lock

- Atomic increment action: $IN_i(A)$

  $\{Read(A);\ A \leftarrow A+k;\ Write(A)\}$

- $IN_i(A)$, $IN_j(A)$ do not conflict!



A=5

$IN_i(A)$ +2

A=7

$IN_j(A)$ +10

A=17

+10 $IN_j(A)$

A=15

+2 $IN_i(A)$

Comp

New request

Lock already held in

	S	X	I
S			
X			
I			

Comp

New request

Lock already held in

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

Update locks

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			

New request

Comp

Lock

Already

held in

	S	X	U
S	T	F	T
X	F	F	F
U	TorF	F	T

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

$$S_1 = \ldots l\text{-}S_1(A) \ldots l\text{-}S_2(A) \ldots l\text{-}U_3(A) \ldots \begin{cases} l\text{-}S_4(A) \ldots ? \\ l\text{-}U_4(A) \ldots ? \end{cases}$$

- 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

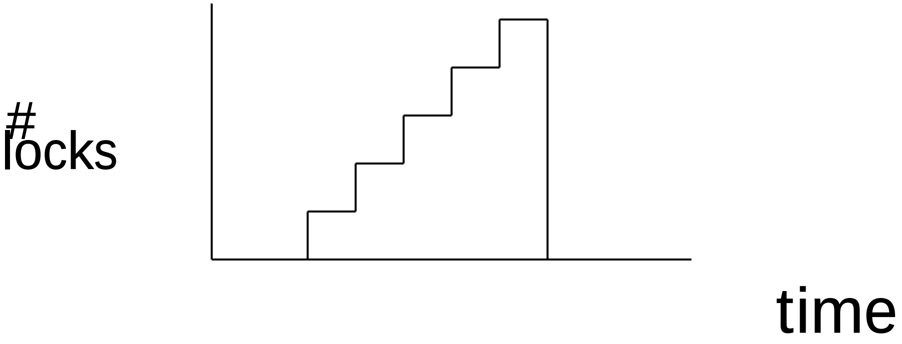
  (E.g., may not even provide
  CONFLICT-SERIALIZABLE schedules)

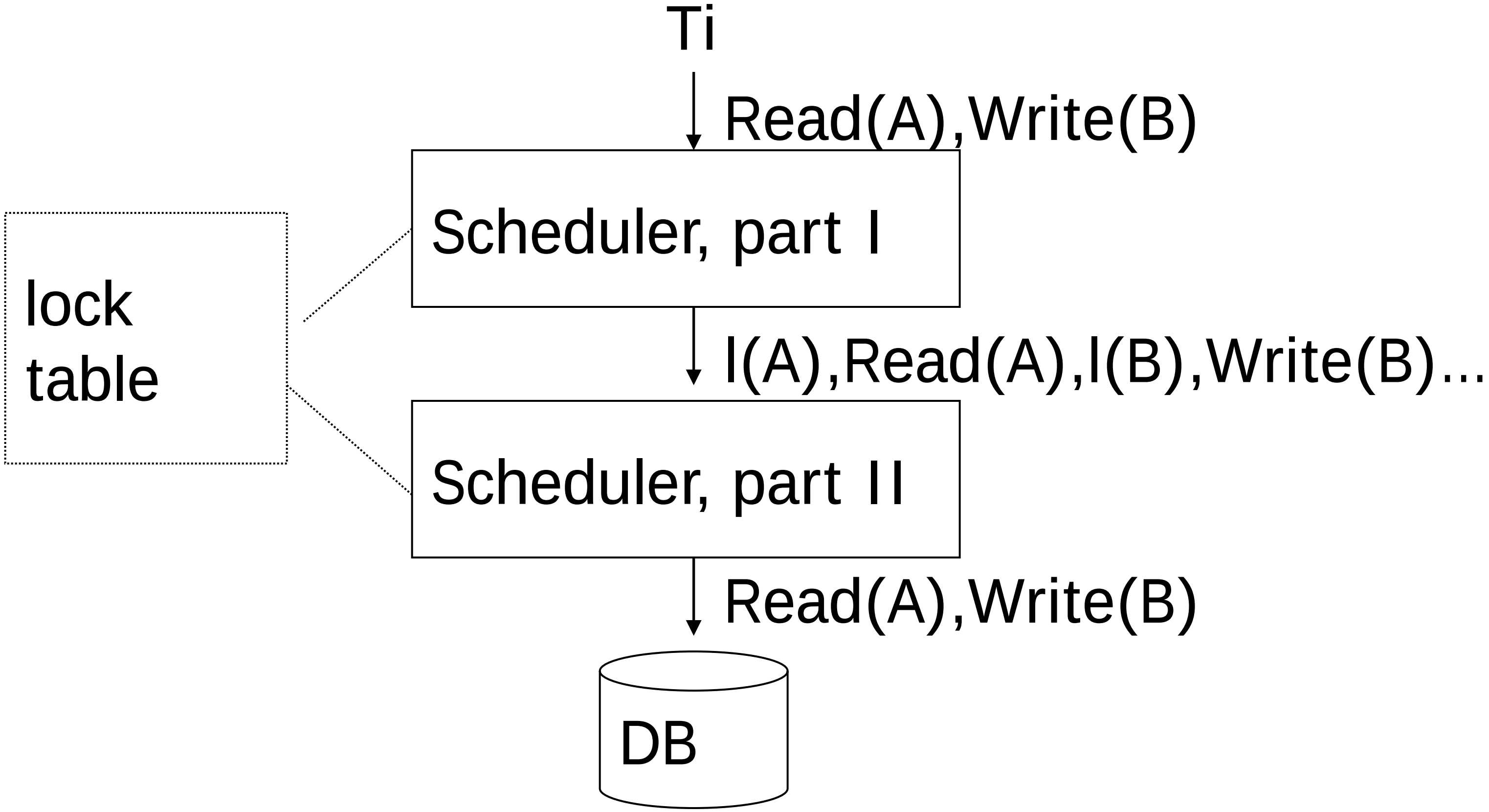
- But here is one (simplified) way ...

Sample Locking System:

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

(2) Hold all locks until transaction commits





Ti

Read(A),Write(B)

Scheduler, part I

lock table

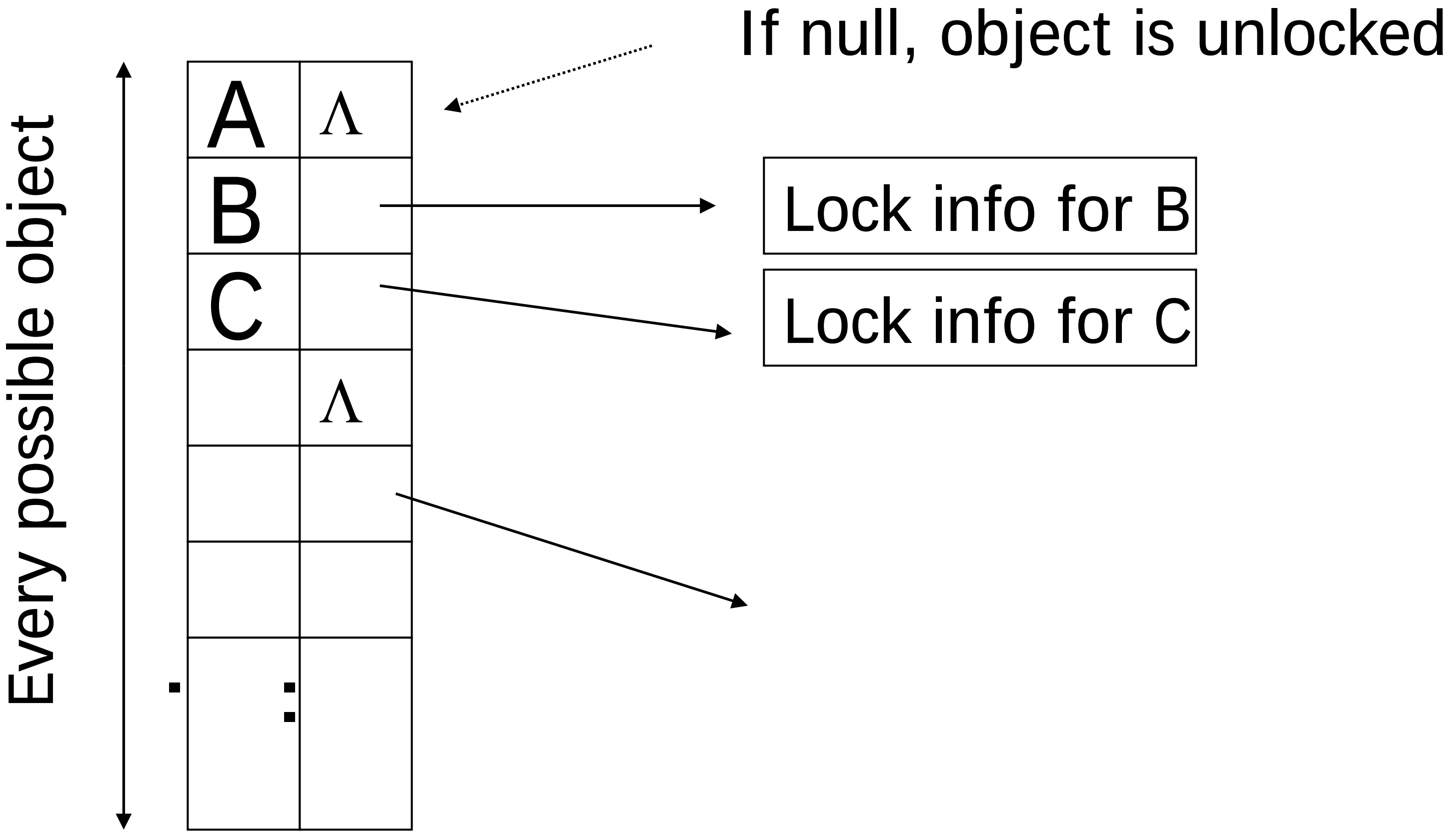
l(A),Read(A),l(B),Write(B)…

Scheduler, part II

Read(A),Write(B)

DB

Lock table   Conceptually



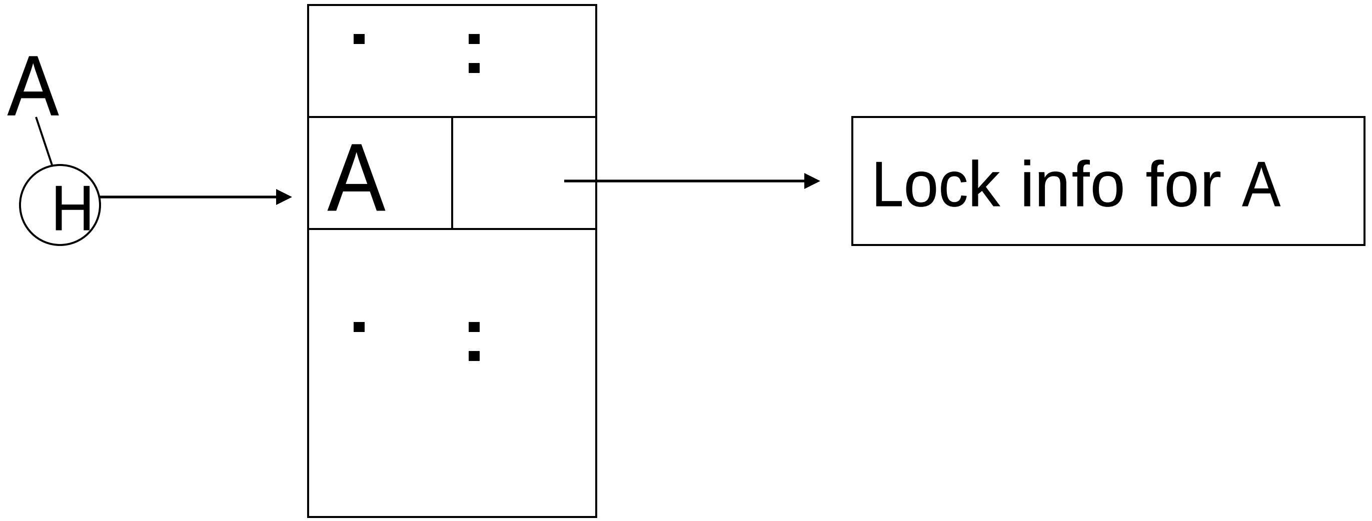
If null, object is unlocked

Lock info for B

Lock info for C

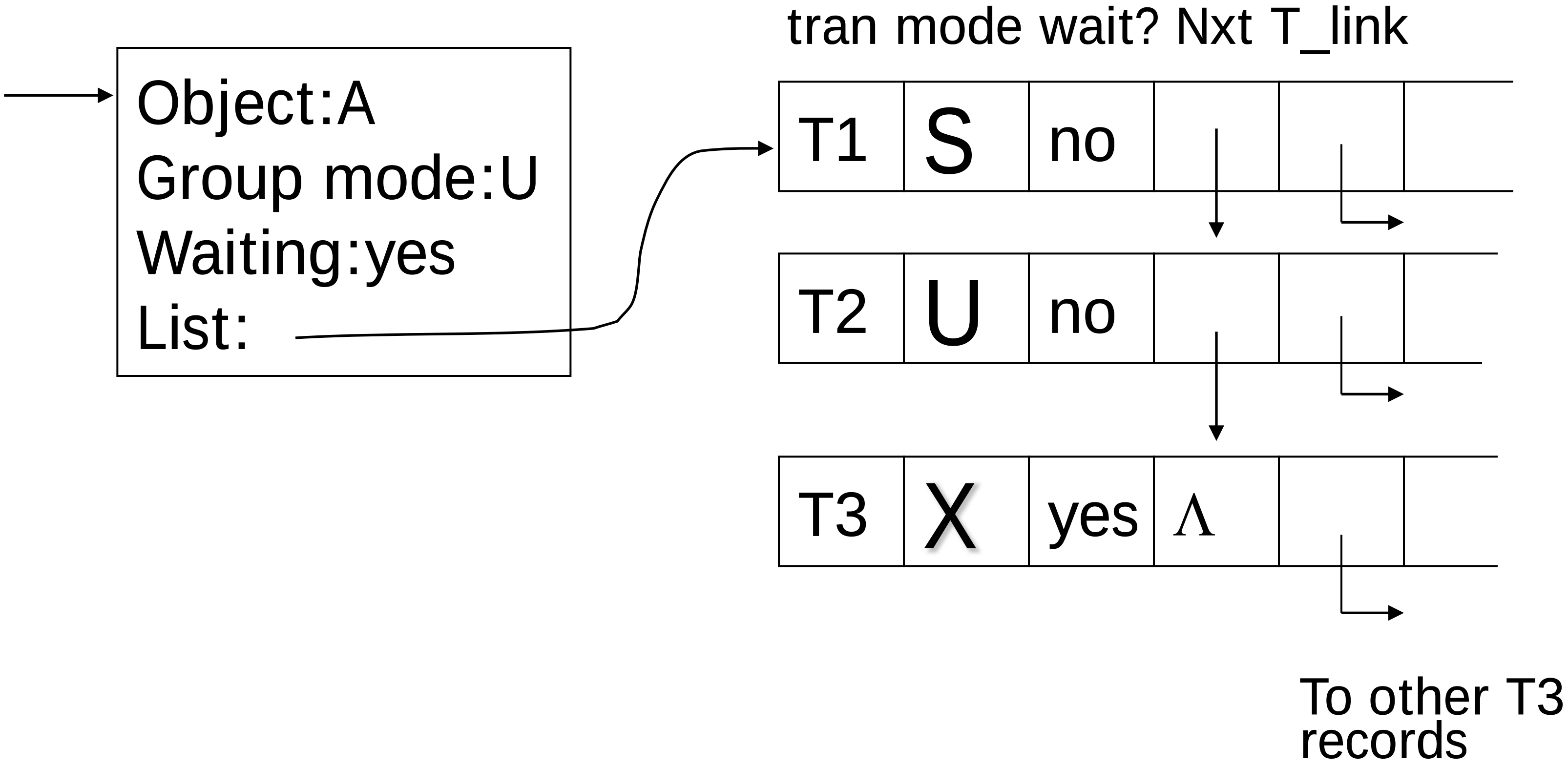
Every possible object

But use hash table:



If object not found in hash table, it is unlocked

Lock info for A - example

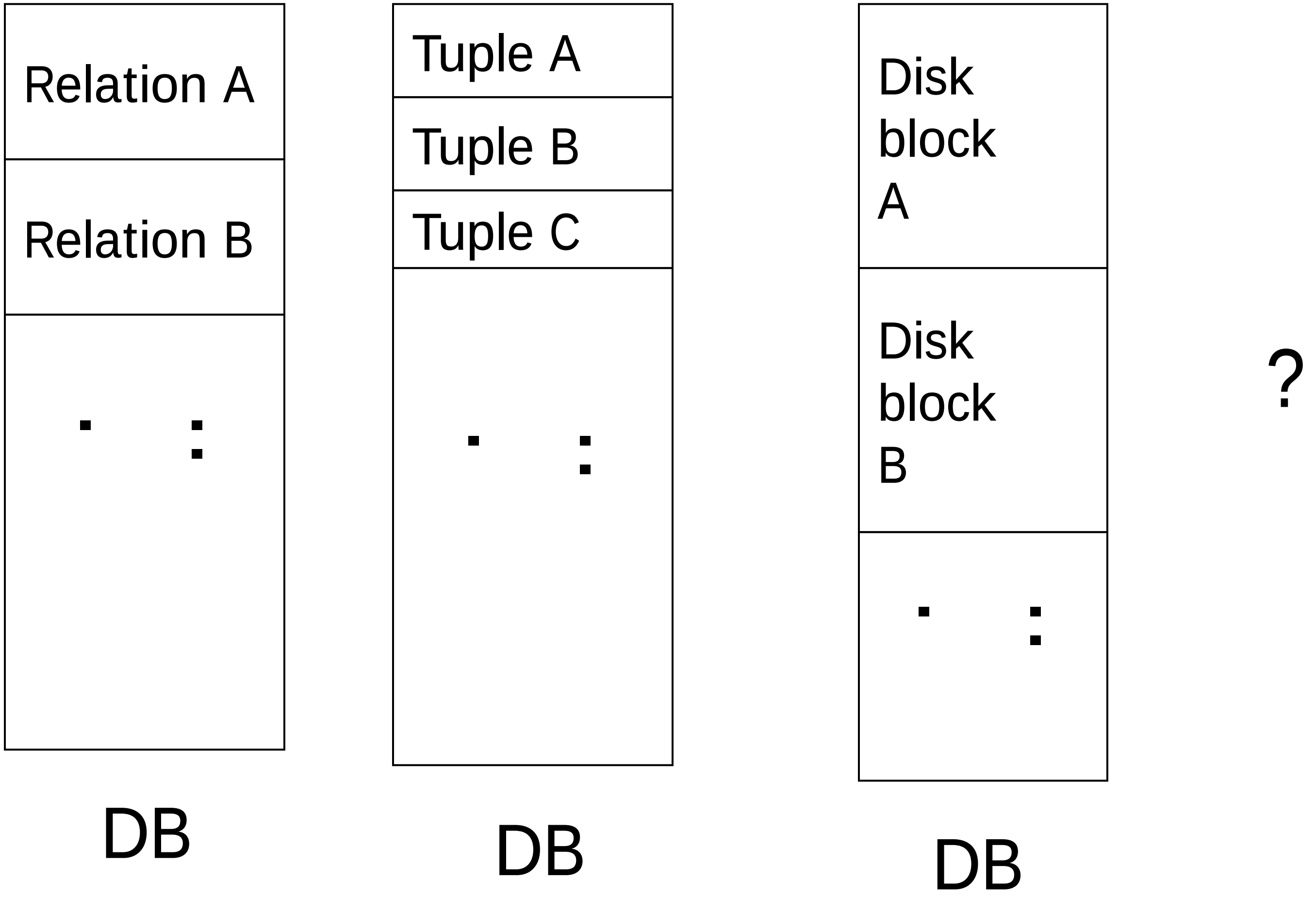


Object:A
Group mode:U
Waiting:yes
List:

tran	mode	wait?	Nxt	T_link
T1	S	no		
T2	U	no		
T3	X	yes	Λ	

To other T3 records

What are the objects we lock?



Relation A

Relation B

. :

DB

Tuple A

Tuple B

Tuple C

. :

DB

Disk block A

Disk block B

. :

DB

?

- 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