Enforcing Serializability
Part 2

Objectives:
- Locking – generalizing from simple locks
  - escalading locks (read/shared vs write/exclusive)
  - update locks
  - multigranular locks (tuple, block, table level locks)
- Soft Isolation in SQL
- How to identify deadlock

Left off Last Time:

- Theorem & proof that 2 phase locking allows only conflict serializable schedules.

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

Proof:

(1) Assume P(S) has cycle

$$T_1 \rightarrow T_2 \rightarrow \ldots T_n \rightarrow T_1$$

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

(3) Impossible, so P(S) acyclic

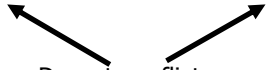
(4) $\Rightarrow$ S is conflict serializable

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



Instead:

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

Lock actions

$l\text{-}t_i(A)$: lock A in t mode (t is S or X)

$u\text{-}t_i(A)$: 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 =... $l\text{-}S_1(A)$ … $r_1(A)$ … $u_1(A)$ …

T_i =... $l\text{-}X_1(A)$ … $w_1(A)$ … $u_1(A)$ …

Locks on same object are now allowed…

but - really :

escalating locks on same object allowed

- What about transactions that read and write same object?

Option 1: Request exclusive lock

T_i = ...$l\text{-}X_1(A)$ … $r_1(A)$ ... $w_1(A)$ ... 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

		request S	request X
held	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)
- beginning of shrinking

Theorem Rules 1,2,3 $\Rightarrow$ Conf.serializable
for S/X locks schedules

Proof: similar to X locks case

Detail:

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)

Compatibility matrix as a proof tool

held \ request	S	X
S	true	false
X	false	false

- The compatibility matrix enables,
  - Methodically enumerate, (each cell), the conflicts.
  - Proof skeleton works, just have to argue correctness of the matrix.

Lock types beyond S/X

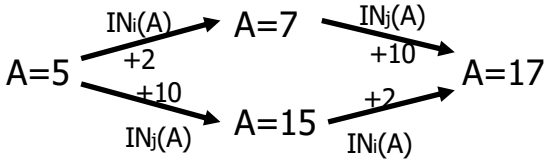
Examples:

(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!



Comp

	S	X	I
S			
X			
I			

Comp

	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)

New request

Comp

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	F	F	F

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

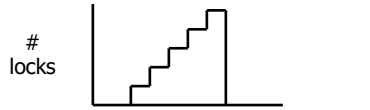
How does locking work in practice?

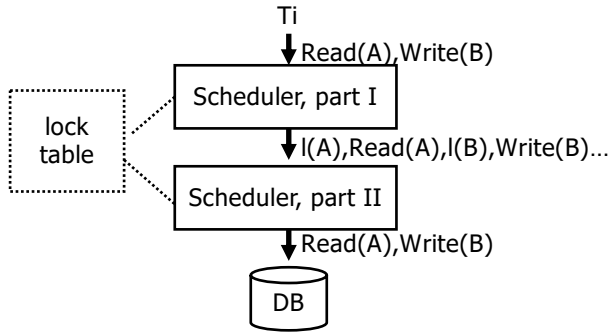
- Every system is different
  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way ...

Sample Locking System:

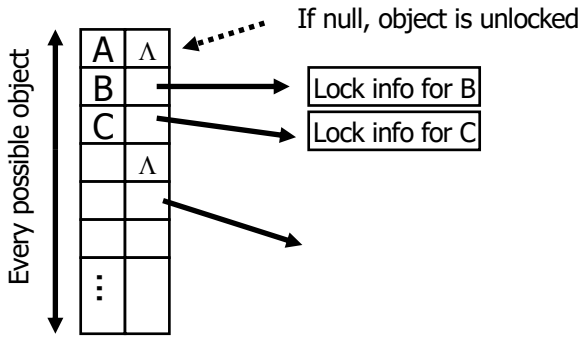
(1) Don't trust transactions to request/release locks

(2) Hold all locks until transaction commits

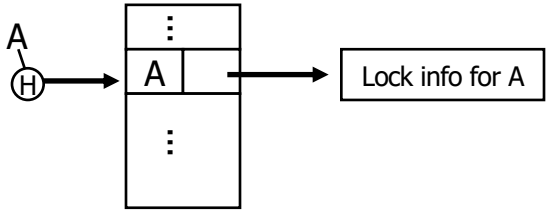




Lock table Conceptually

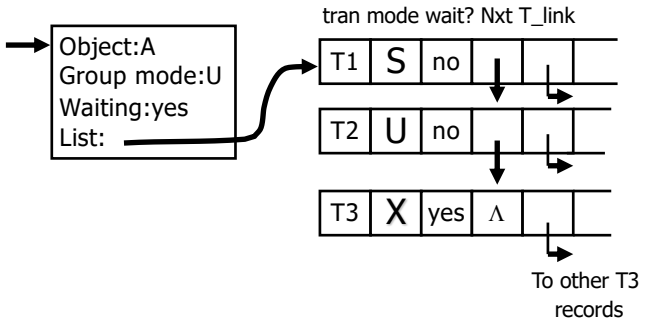


But use hash table:

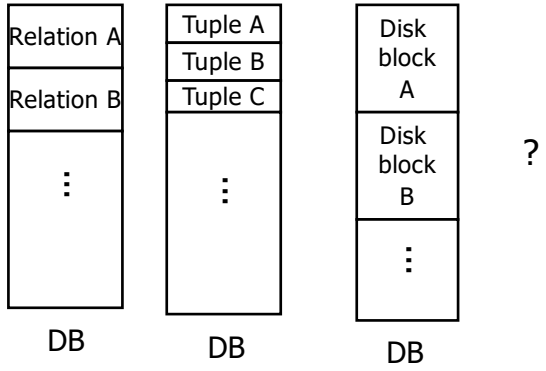


If object not found in hash table, it is unlocked

Lock info for A - example



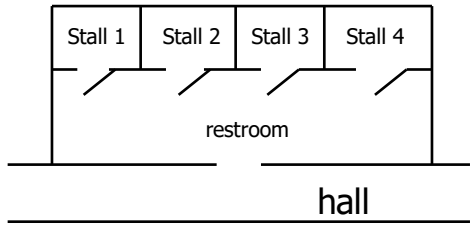
What are the objects we lock?



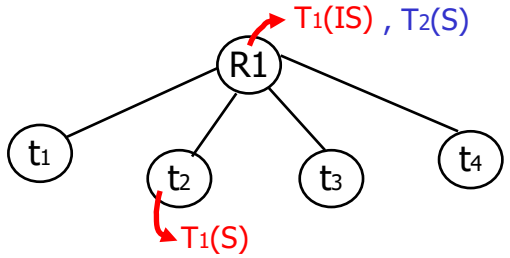
- Locking works in any case, but should we choose small or large objects?
- If we lock large objects (e.g., Relations)
  - Need few locks
  - Low concurrency
- If we lock small objects (e.g., tuples,fields)
  - Need more locks
  - More concurrency

We can have it both ways!!

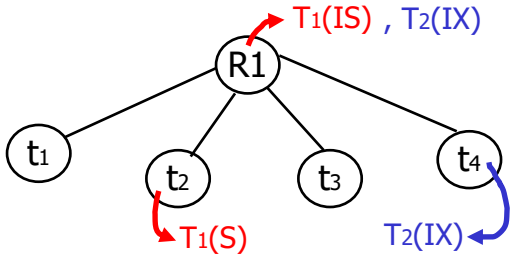
Ask any janitor to give you the solution...



Example



Example



Multiple granularity

Comp

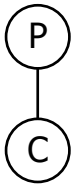
Holder \ Requestor	IS	IX	S	SIX	X
IS					
IX					
S					
SIX					
X					

Multiple granularity

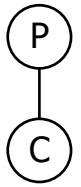
Comp

Holder \ Requestor	IS	IX	S	X
IS	T	T	T	F
IX	T	T	F	F
S	T	F	T	F
X	F	F	F	F

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



Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X
S	[S, IS] not necessary
X	none

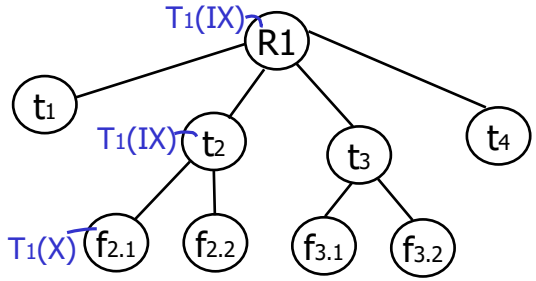


Rules

(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,IX only if parent(Q) locked by Ti in IX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q' s children are locked by Ti

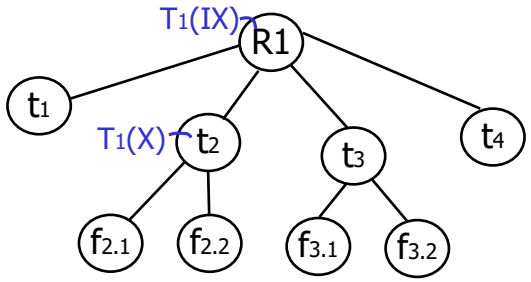
Exercise:

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



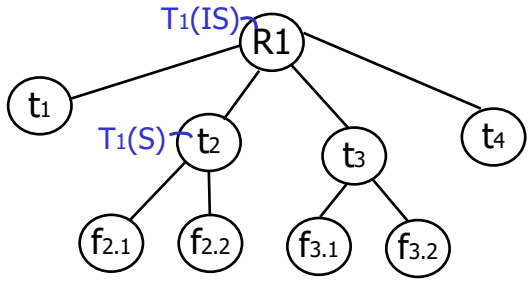
Exercise:

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



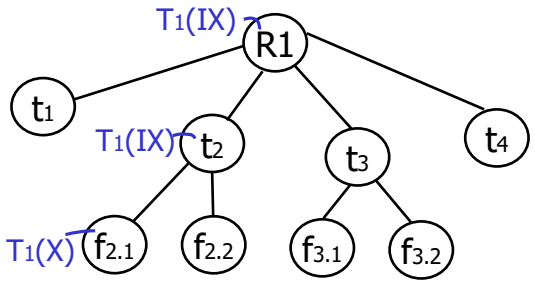
Exercise:

- 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?



SQL Isolation Levels

Hard and Soft Transactions in RDBMS

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- Only one level ("serializable") = ACID transactions.
- Each DBMS implements transactions in its own way.

Choosing the Isolation Level

- Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

What About Deadlock?

Waits-for-Graph

- Used for dead lock detection.
- Intuitively similar to a precedence graph

But

- edges are populated dynamically.
- Instead of conflicts, edge, (Ti, Tj) exists if Ti is blocked from running by a lock held by Tj
- A cycle in the graph → transactions are deadlocked.

Waits-for-Graph

- Need a daemon to periodically wake up and check the waits-for-graph for cycles.
- How often?

  A tradeoff

  - For a large number of transactions, can be expensive.
    → Less often.
  - If there is a deadlock, transactions (customers) are waiting.
    →More often