Transactions & Concurrency Control II

Alvin Cheung
Aditya Parameswaran

Reading: R & G Chapter 16-17

Transaction Implementations

- Many available!
  - Targets different workloads
- We will focus on **lock-based** implementations
  - Others: use multiple versions of data and “optimistically” let transactions move forward
    - Abort when conflicts are detected
  - Some names to know/look up:
    - Optimistic Concurrency Control
    - Timestamp-Ordered Multiversion Concurrency Control
  - We will not study these schemes in this lecture

“Lock” data??

- Not by any crypto or hardware enforcement
  - There are no adversaries here … this is all within the DBMS
- Recall locks / semaphores from 61c
  - These are synchronization primitives
  - Locking / unlocking has *costs*
- We lock by simple convention within the DBMS:
  - Each *data element* has a unique lock
  - Each transaction must first acquire the lock before reading/writing that element
  - If the lock is taken by another transaction, then wait
  - The transaction must release the lock(s) at some point
- Different *lock protocols / schemes* differ by:
  - When to lock / unlock each data element
  - What data element to lock
  - What happens when a txn waits for a lock

What are “data elements”?

Major differences between vendors:

- Lock on the entire database
  - SQLite
- Lock on individual records
  - SQL Server, DB2, etc
- Will see tradeoffs later on

Actions on Locks

$Lock_i(A)$ / $L_i(A)$ = transaction T_i acquires lock for element A

$Unlock_i(A)$ / $U_i(A)$ = transaction T_i releases lock for element A

Let's see this in action…

A Non-Serializable Schedule

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)	

Example

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Using locks has ensured a conflict-serializable schedule

But…

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$;	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;
$L_1(B)$; READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	

Locks did not enforce conflict-serializability!!! What’s wrong ?

Two Phase Locking (2PL)

Berkeley cs186

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests

Example: 2PL transactions

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;

Now it is conflict-serializable

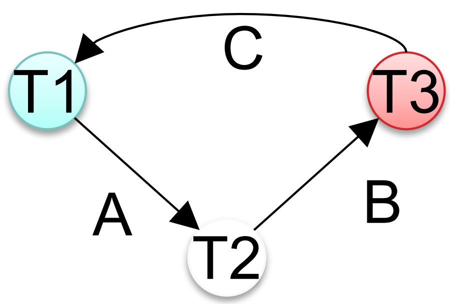
Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

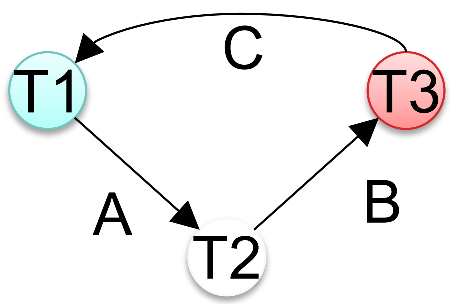
Proof. Suppose not: then there exists a cycle in the dependence graph.



Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the dependence graph.

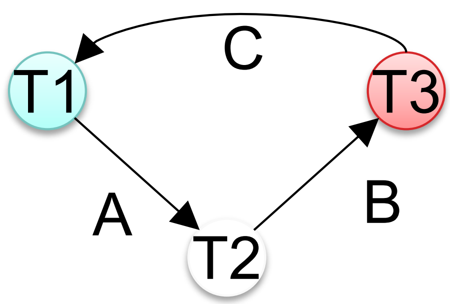


Then there is the following **temporal** cycle in the schedule:

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the dependence graph.



Then there is the following **temporal** cycle in the schedule:

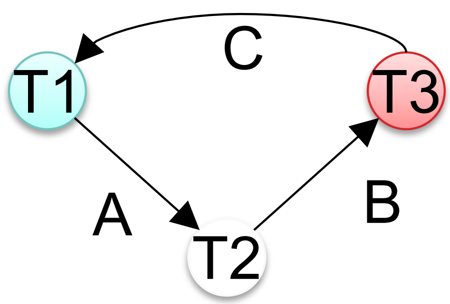
$U_1(A) \rightarrow L_2(A)$ why?

$U_1(A)$ happened strictly *before* $L_2(A)$

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the dependence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

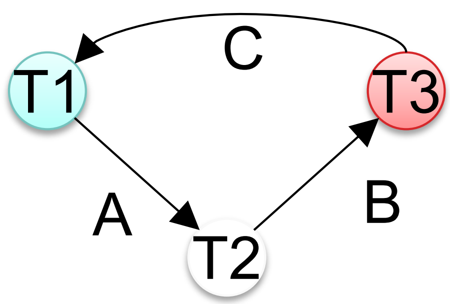
$L_2(A) \rightarrow U_2(B)$ why?

$L_2(A)$ happened strictly *before* $U_1(A)$

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the dependence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

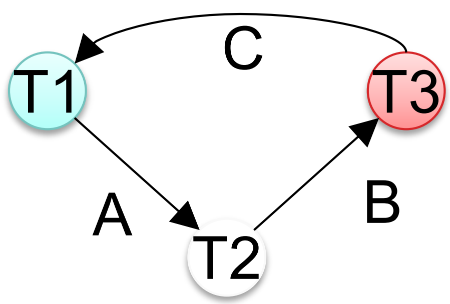
$L_2(A) \rightarrow U_2(B)$

$U_2(B) \rightarrow L_3(B)$ why?

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the dependence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$

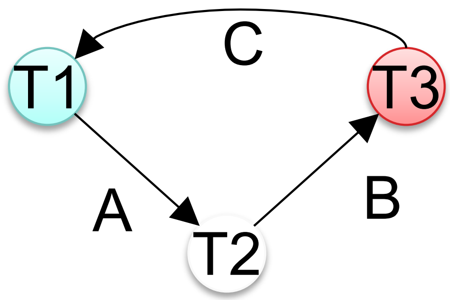
$U_2(B) \rightarrow L_3(B)$

......etc.....

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the dependence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$

$U_2(B) \rightarrow L_3(B)$

$L_3(B) \rightarrow U_3(C)$

$U_3(C) \rightarrow L_1(C)$

$L_1(C) \rightarrow U_1(A)$

Cycle in time: Contradiction

A New Problem: Non-recoverable Schedule

T1	T2
L_1(A); L_1(B); READ(A)	
A :=A+100	
WRITE(A); U_1(A)	
	L_2(A); READ(A)
	A := A*2
	WRITE(A);
	L_2(B); BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); U_1(B);	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); U_2(A); U_2(B);
	Commit
Rollback	

Aka cascading aborts

A New Problem: Non-recoverable Schedule

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

Elements A, B written by T1 are restored to their original value.

A New Problem: Non-recoverable Schedule

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A :=A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;
	Commit
Rollback	

Dirty reads of A, B lead to incorrect writes.

Elements A, B written by T1 are restored to their original value.

A New Problem: Non-recoverable Schedule

T1	T2
L_1(A); L_1(B); READ(A)	
A :=A+100	
WRITE(A); U_1(A)	
	L_2(A); READ(A)
	A := A*2
	WRITE(A);
	L_2(B); BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); U_1(B);	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); U_2(A); U_2(B);
	Commit
Rollback	

Dirty reads of A, B lead to incorrect writes.

Elements A, B written by T1 are restored to their original value.

Can no longer undo!

Strict 2PL

The Strict 2PL rule:

All locks are held until commit/abort: All unlocks are done together with commit/abort.

With strict 2PL, we will get schedules that are both conflict-serializable and recoverable

Strict 2PL

T1	T2
L_1(A); READ(A)	
A :=A+100	
WRITE(A);	
	L_2(A); BLOCKED…
L_1(B); READ(B)	
B :=B+100	
WRITE(B);	
Rollback & U_1(A);U_1(B);	
	…GRANTED; READ(A)
	A := A*2
	WRITE(A);
	L_2(B); READ(B)
	B := B*2
	WRITE(B);
	Commit & U_2(A); U_2(B);

Strict 2PL

- Lock-based systems always use strict 2PL
- Easy to implement:
  - Before a transaction reads or writes an element A, insert an L(A)
  - When the transaction commits/aborts, then release all locks
- Ensures both conflict serializability and recoverability

Another problem: Deadlocks

- T_1: R(A), W(B)
- T_2: R(B), W(A)

- T_1 holds the lock on A, waits for B
- T_2 holds the lock on B, waits for A

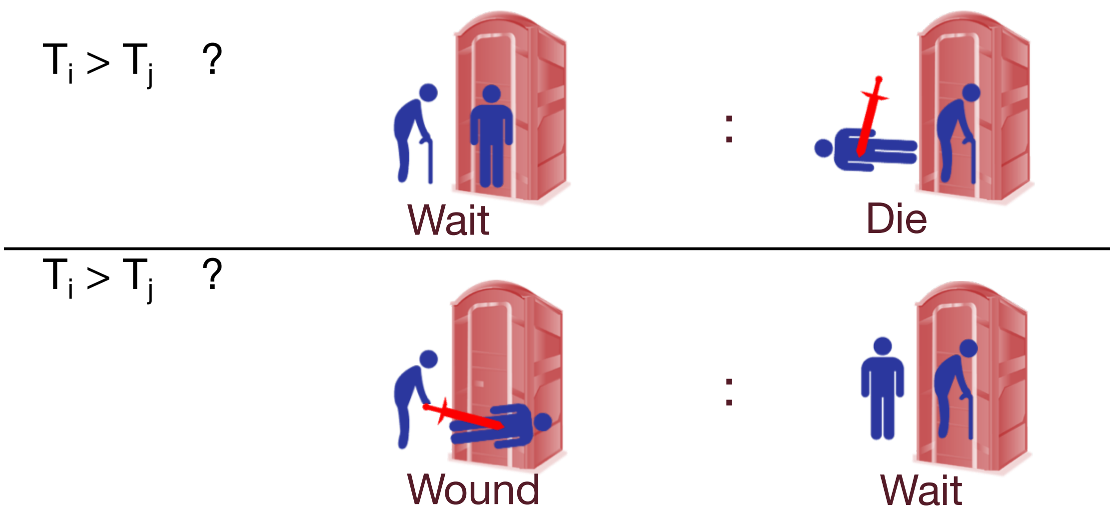
This is a deadlock!

Deadlock Prevention

- **Common technique in operating systems**
- **Standard approach: resource ordering**
  - Screen < Network Card < Printer
- **Why is this problematic for transactions?**
  - What order would you impose?

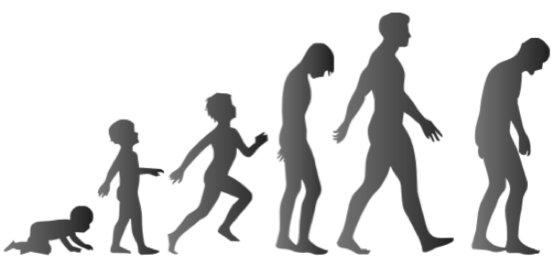
Deadlock Avoidance

- **Assign priorities based on age: (now – start_time).**
- Say T_i wants a lock that T_j holds. Two possible policies:
  - **Wait-Die:** If T_i has higher priority, T_i waits for T_j; else T_i aborts
  - **Wound-Wait:** If T_i has higher priority, T_j aborts; else T_i waits
  - Read each of these like a ternary operator (C/C++/java/javascript)



Deadlock Avoidance: Analysis

- Q: Why do these schemes guarantee no deadlocks?
  - Q: What do the previous images have in common?
- Important Detail: If a transaction re-starts, make sure it gets its original timestamp. Why?
- Note: other priority schemes make sense
  - E.g. measures of resource consumption, like #locks acquired



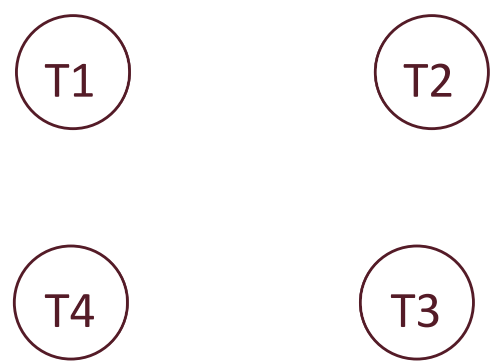
Deadlock Detection

- Create and maintain a **“waits-for” graph**
- Periodically check for cycles in a graph

Deadlock Detection, Part 2

Example:

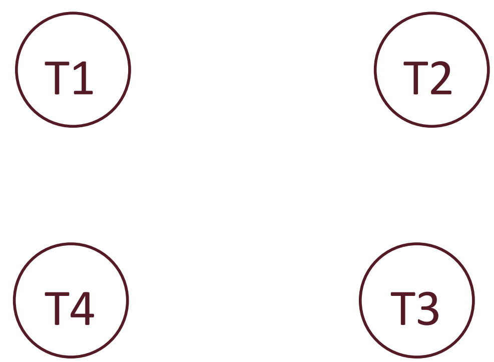
T1:
T2:
T3:
T4:



Deadlock Detection, Part 3

Example:

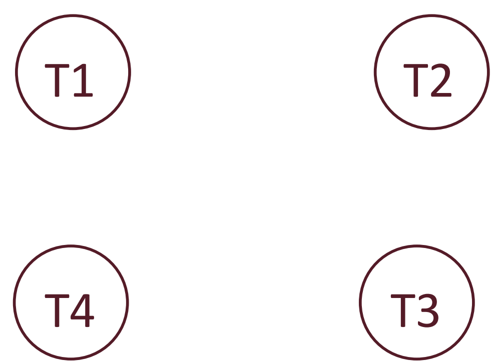
T1: R(A)
T2:
T3:
T4:



Deadlock Detection, Part 4

Example:

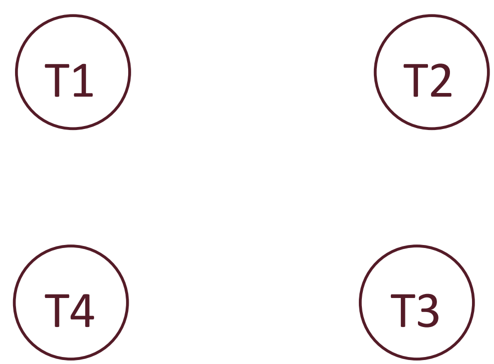
T1: R(A) R(D)
T2:
T3:
T4:



Deadlock Detection, Part 5

Example:

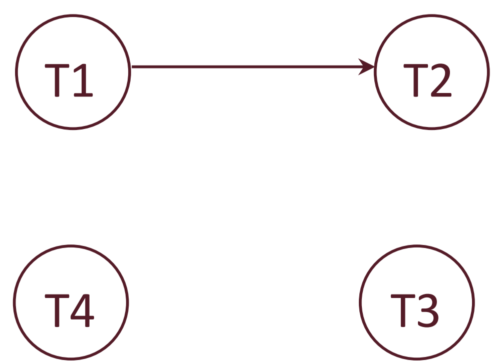
T1: R(A) R(D)
T2: W(B)
T3:
T4:



Deadlock Detection, Part 6

Example:

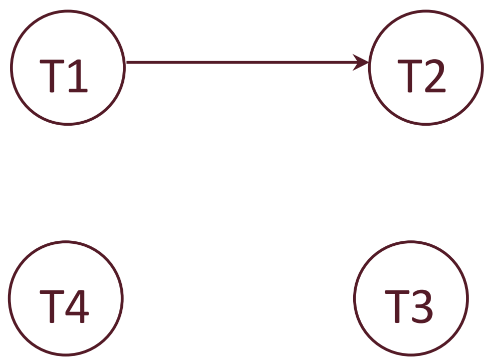
T1: R(A) R(D) R(B)
T2: W(B)
T3:
T4:



Deadlock Detection, Part 7

Example:

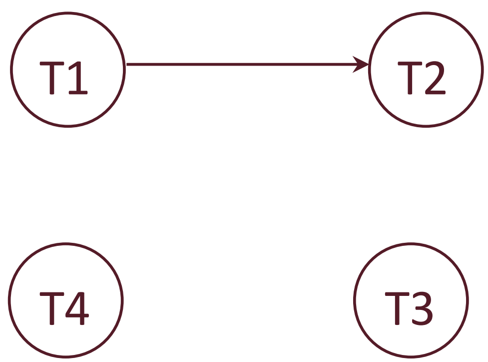
T1:	**R(A)**	**R(D)**		**R(B)**	
T2:			**W(B)**		
T3:					**R(D)**
T4:					



Deadlock Detection, Part 8

Example:

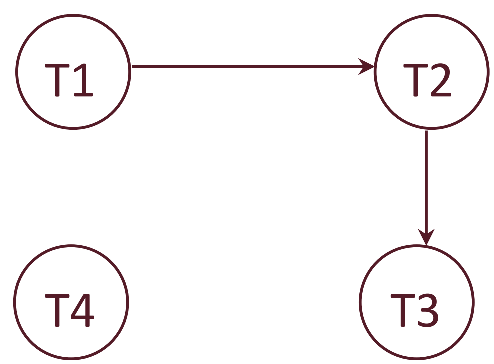
T1:	**R(A)**	**R(D)**		**R(B)**		
T2:			**W(B)**			
T3:					**R(D)**	**R(C)**
T4:						



Deadlock Detection, Part 9

Example:

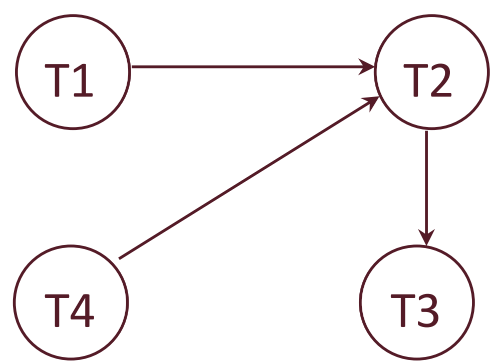
T1:	**R(A)**	**R(D)**		**R(B)**			
T2:			**W(B)**				**W(C)**
T3:					**R(D)**	**R(C)**	
T4:							



Deadlock Detection, Part 10

Example:

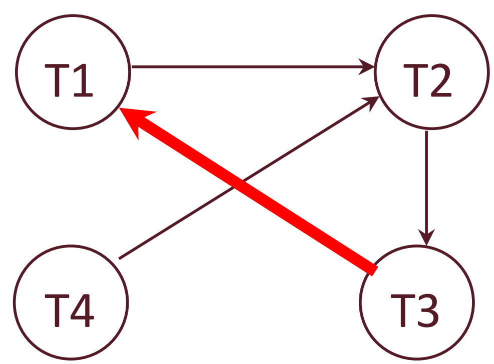
T1:	**R(A)**	**R(D)**		**R(B)**				
T2:			**W(B)**				**W(C)**	
T3:					**R(D)**	**R(C)**		
T4:								**W(B)**



Deadlock Detection, Part 11

Example:

T1:	**R(A)**	**R(D)**		**R(B)**					
T2:			**W(B)**				**W(C)**		
T3:					**R(D)**	**R(C)**			**W(A)**
T4:								**W(B)**	



Deadlock!

- T1, T2, T3 are deadlocked
  - Doing no good, and holding locks
- T4 still cruising
- In the background, run a deadlock detection algorithm
  - Periodically extract the waits-for graph
  - Find cycles
  - “Shoot” a transaction on the cycle
- Empirical fact
  - Most deadlock cycles are small (2-3 transactions)

Lock Modes

Berkeley cs186

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)
- Cannot get new locks after releasing any locks (strict 2PL)

Lock compatibility matrix:

	None	S	X
None			
S			
X			

Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)
- Cannot get new locks after releasing any locks (strict 2PL)

Lock compatibility matrix:

	None	S	X
None	✔	✔	✔
S	✔	✔	✖
X	✔	✖	✖

Lock Management

- Lock and unlock requests handled by Lock Manager
- LM maintains a hashtable, keyed on names of objects being locked.
- LM keeps an entry for each currently held lock
- Entry contains
  - Granted set: Set of txns currently granted access to the lock
  - Lock mode: Type of lock held (shared or exclusive)
  - Wait Queue: Queue of lock requests

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

Lock Management (continued)

- **When lock request arrives:**
  - Does any txn in Granted Set or Wait Queue want a conflicting lock?
    - If no, put the requester into "granted set" and let them proceed
    - If yes, put requester into wait queue (typically FIFO)

- **Lock upgrade:**
  - Txn with shared lock can request to upgrade to exclusive

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
  - E.g., SQL Server
- Coarse grain locking (e.g., tables, entire database)
  - Many false conflicts
  - Less overhead in managing locks
  - E.g., SQL Lite
- Solution: lock escalation changes granularity as needed

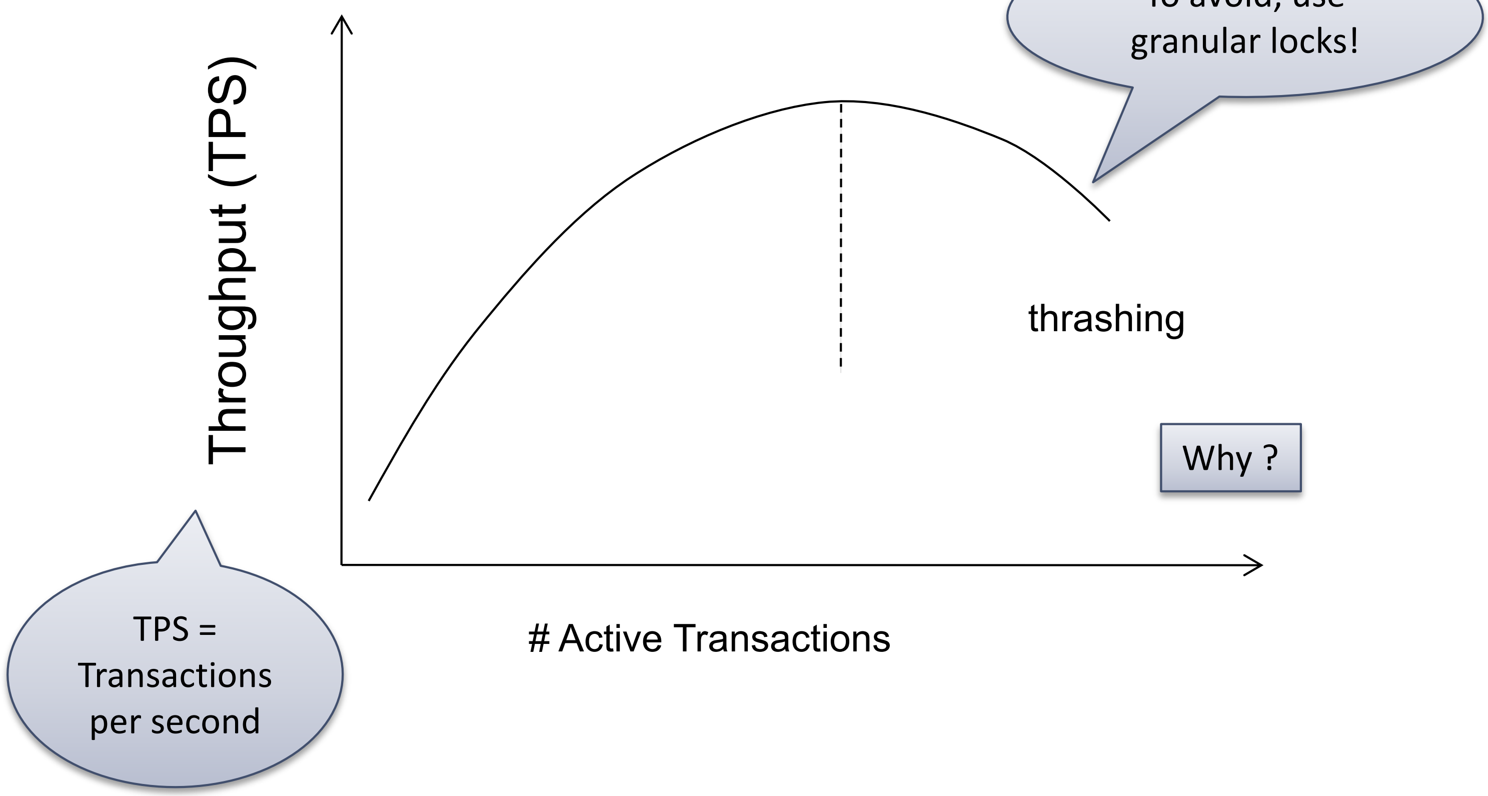
Lock Granularity, cont

- Hard to decide what granularity to lock
  - Tuples vs Pages vs Tables?
- What is the tradeoff?
  - Fine-grained availability of resources would be nice (e.g. lock per tuple)
  - Small # of locks to manage would also be nice (e.g. lock per table)
  - Can't have both!
    - Or can we???

Multiple Locking Granularity

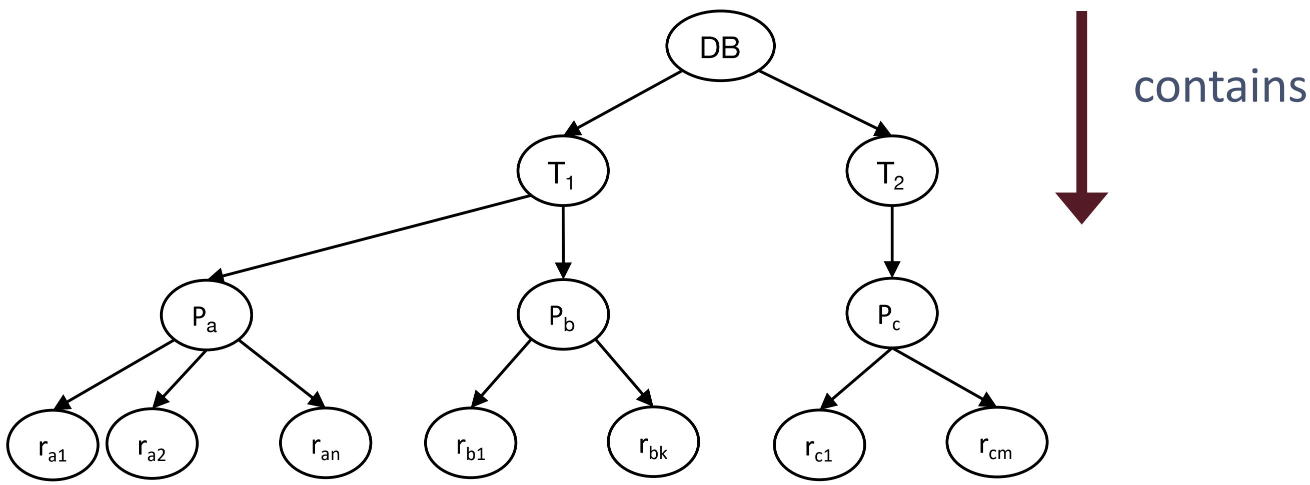
- **Shouldn't have to make same decision for all transactions!**
- Allow data items to be of various sizes
- Define a hierarchy of data granularities, small nested within large
  - Can be represented graphically as a tree.

Lock Performance



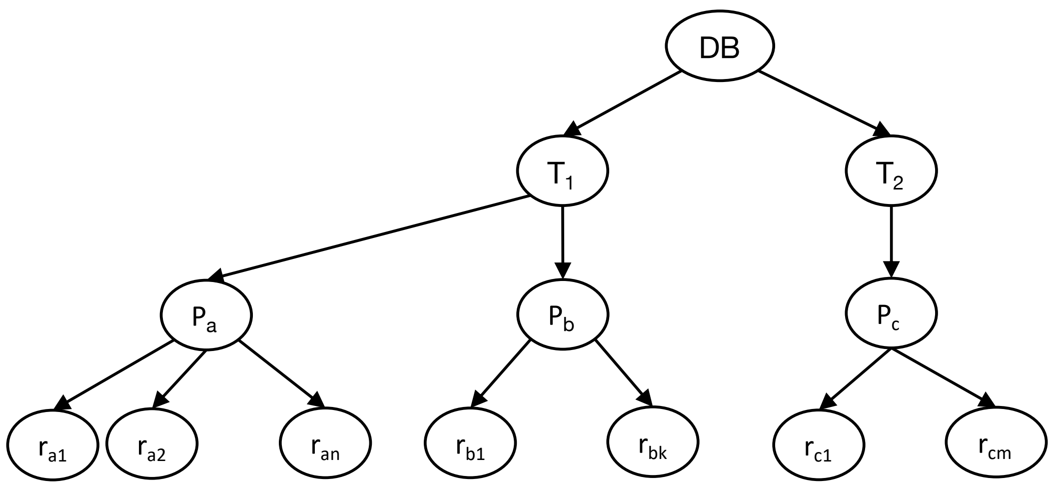
Example of Granularity Hierarchy

- Data "containers" can be viewed as nested.
- The levels, starting from the coarsest (top) level are
  - Database, Tables, Pages, Records
- When a transaction locks a node in the tree ***explicitly***, it ***implicitly*** locks all the node's descendants in the same mode.



Multiple Locking Granularity

- Granularity of locking (level in tree where locking is done):
  - **Fine granularity** (lower in tree): High concurrency, lots of locks (overhead)
  - **Coarse granularity** (higher in tree): Few locks (low overhead), lost concurrency
    - Lost potential concurrency if you don't need everything inside the coarse grain



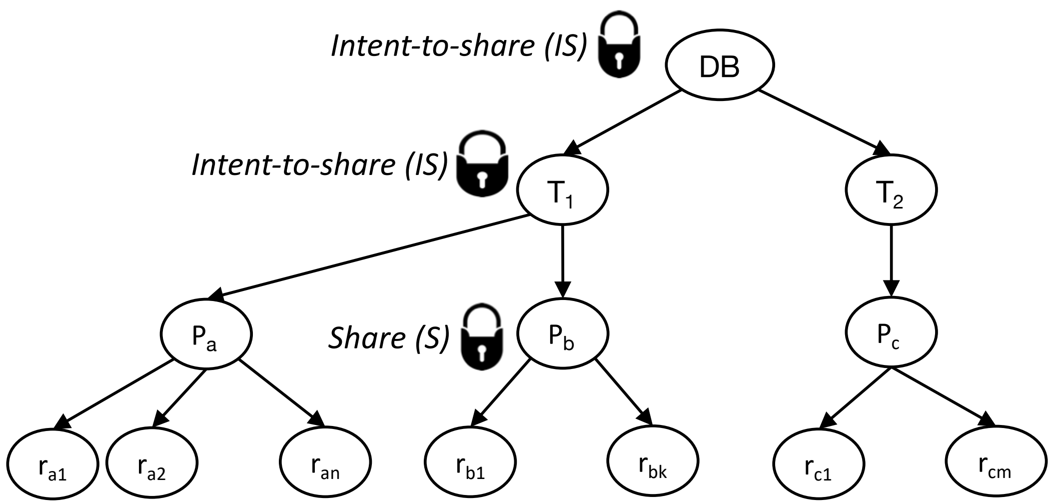
Real-World Locking Granularities

Resource	Description
RID	A row identifier used to lock a single row within a heap.
KEY	A row lock within an index used to protect key ranges in serializable transactions.
PAGE	An 8-kilobyte (KB) page in a database, such as data or index pages.
EXTENT	A contiguous group of eight pages, such as data or index pages.
HoBT	A heap or B-tree. A lock protecting a B-tree (index) or the heap data pages in a table that does not have a clustered index.
TABLE	The entire table, including all data and indexes.
FILE	A database file.
APPLICATION	An application-specified resource.
METADATA	Metadata locks.
ALLOCATION_UNIT	An allocation unit.
DATABASE	The entire database.

From MS SQL Server
https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx

New Lock Modes and Protocol

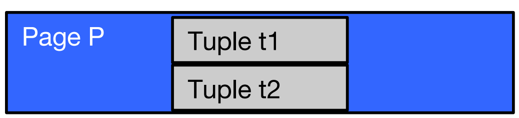
- Allow txns to lock at each level, but with a special protocol using new "intent" locks:
- Before getting S or X lock, txn must have proper intent locks on all its ancestors in the granularity hierarchy.



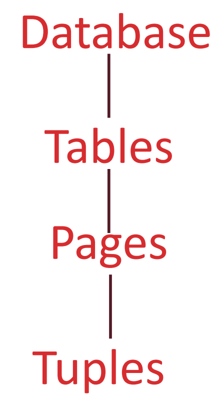
New Lock Modes – Intention Lock Modes

- 3 additional lock modes:
  - ***IS:*** *Intent to get S lock(s) at finer granularity.*
  - ***IX:*** *Intent to get X lock(s) at finer granularity.*
  - ***SIX:*** *Like S & IX at the same time. Why useful?*

- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes



Multiple Granularity Locking Protocol



- Each txn starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

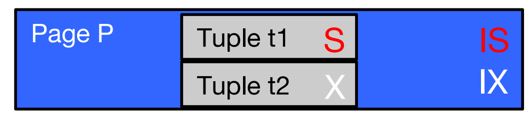
- Enforce (strict) 2-phase locking as before
- Protocol is correct in that it is *equivalent to directly setting locks at leaf levels of the hierarchy*.

- What does the lock compatibility matrix look like?

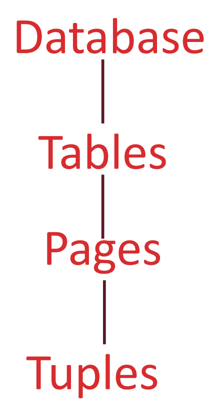
Lock Compatibility Matrix

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

Handy simple case to remember:
Could 2 intent locks be compatible?



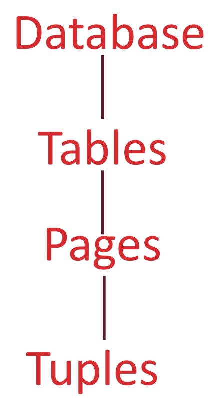
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX		true	false	false	false
S			true	false	false
SIX				false	false
X			false		false



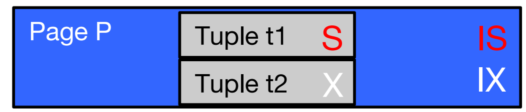
Lock Compatibility Matrix, Cont

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false



Handy simple case to remember:
Could 2 intent locks be compatible?



Real-World Lock Compatibility Matrix

Berkeley cs186

	NL	SCH-S	SCH-M	S	U	X	IS	IU	IX	SIU	SIX	UIX	BU	RS-S	RS-U	RI-N	RI-S	RI-U	RI-X	RX-S	RX-U	RX-X
NL	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
SCH-S	N	N	C	N	N	N	N	N	N	N	N	N	N	I	I	I	I	I	I	I	I	I
SCH-M	N	C	C	C	C	C	C	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
S	N	N	C	N	N	C	N	N	C	N	C	C	C	N	N	N	N	N	C	N	N	C
U	N	N	C	N	C	C	N	C	C	C	C	C	C	N	C	N	N	C	C	N	C	C
X	N	N	C	C	C	C	C	C	C	C	C	C	C	C	C	N	C	C	C	C	C	C
IS	N	N	C	N	N	C	N	N	N	N	N	N	C	I	I	I	I	I	I	I	I	I
IU	N	N	C	N	C	C	N	N	N	N	N	C	C	I	I	I	I	I	I	I	I	I
IX	N	N	C	C	C	C	N	N	N	C	C	C	C	I	I	I	I	I	I	I	I	I
SIU	N	N	C	N	C	C	N	N	C	N	C	C	C	I	I	I	I	I	I	I	I	I
SIX	N	N	C	C	C	C	N	N	C	C	C	C	C	I	I	I	I	I	I	I	I	I
UIX	N	N	C	C	C	C	N	C	C	C	C	C	C	I	I	I	I	I	I	I	I	I
BU	N	N	C	C	C	C	C	C	C	C	C	C	N	I	I	I	I	I	I	I	I	I
RS-S	N	I	I	N	N	C	I	I	I	I	I	I	I	N	N	C	C	C	C	C	C	C
RS-U	N	I	I	N	C	C	I	I	I	I	I	I	I	N	C	C	C	C	C	C	C	C
RI-N	N	I	I	N	N	N	I	I	I	I	I	I	I	C	C	N	N	N	N	C	C	C
RI-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	N	N	N	C	C	C	C
RI-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	N	N	C	C	C	C	C
RI-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	N	C	C	C	C	C	C
RX-S	N	I	I	N	N	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-U	N	I	I	N	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C
RX-X	N	I	I	C	C	C	I	I	I	I	I	I	I	C	C	C	C	C	C	C	C	C

Key

N	No Conflict	SIU	Share with Intent Update
I	Illegal	SIX	Shared with Intent Exclusive
C	Conflict	UIX	Update with Intent Exclusive
		BU	Bulk Update
NL	No Lock	RS-S	Shared Range-Shared
SCH-S	Schema Stability Locks	RS-U	Shared Range-Update
SCH-M	Schema Modification Locks	RI-N	Insert Range-Null
S	Shared	RI-S	Insert Range-Shared
U	Update	RI-U	Insert Range-Update
X	Exclusive	RI-X	Insert Range-Exclusive
IS	Intent Shared	RX-S	Exclusive Range-Shared
IU	Intent Update	RX-U	Exclusive Range-Update
IX	Intent Exclusive	RX-X	Exclusive Range-Exclusive

From MS SQL Server
https://technet.microsoft.com/en-us/library/jj856598(v=sql.110).aspx

Phantom Problem

- So far we have assumed the database to be a *static* collection of elements (=tuples)

- If tuples are inserted/deleted then the *phantom problem* appears

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

Is this schedule serializable ?

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘A3’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Is this schedule serializable ?

No: T1 sees a “phantom” product A3

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

But this is conflict-serializable!

$R_1(A1);R_1(A2);W_2(A3);R_1(A1);R_1(A2);R_1(A3)$

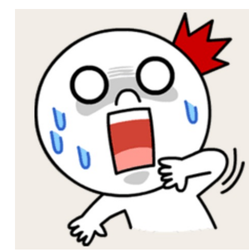
$W_2(A3);R_1(A1);R_1(A2);R_1(A1);R_1(A2);R_1(A3)$

Phantom Problem

Berkeley cs186

- A "phantom" is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution

- In our example:
  - T1: reads list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears !



- Conflict-serializability assumes DB is *static*
- When DB is *dynamic* then c-s is not serializable.

Dealing With Phantoms

- Lock the entire table
- Lock the index entry for ‘blue’
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

Dealing with phantoms is expensive !

Summary of Serializability

- Serializable schedule = equivalent to a serial schedule
- (strict) 2PL guarantees *conflict serializability*
  - What is the difference?
- **Static database**:
  - *Conflict serializability* implies serializability
- **Dynamic database**:
  - This no longer holds

Summary, cont.

- **Correctness criterion for isolation is "serializability".**
  - In practice, we use "conflict serializability" which is conservative but easy to enforce
- **Two Phase Locking and Strict 2PL: Locks implement the notions of conflict directly**
  - The lock manager keeps track of the locks issued.
  - **Deadlocks** may arise; can either be prevented or detected.
- **Multi-Granularity Locking:**
  - Allows flexible tradeoff between lock "scope" in DB, and # of lock entries in lock table
- **More to the story**
  - Optimistic/Multi-version/Timestamp CC
  - Index "latching", phantoms
  - Actually, there's much much more ☺