



Chapter 15 : Concurrency Control

CSR IS TO COSTA FOR
PRACTICAL USE - HERE, WE
ARE GOING TO DISCUSS SOME
ALTERNATIVES BASED ON LOCKS

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
AN ENTIRE TABLE, A SET OF ITS COLUMNS, ...
- Data items can be locked in two modes :
 1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
 2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.



Lock-Based Protocols (Cont.)

■ Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
 - But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.



Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

```
 $T_2$ : lock-S(A);  
read (A);  
unlock(A);  
lock-S(B);  
read (B);  
unlock(B);  
display(A+B)
```

- Locking as above is not sufficient to guarantee serializability
 - if A and B get updated in-between the read of A and B , the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.



The Two-Phase Locking Protocol

- This protocol ensures conflict-serializable schedules.

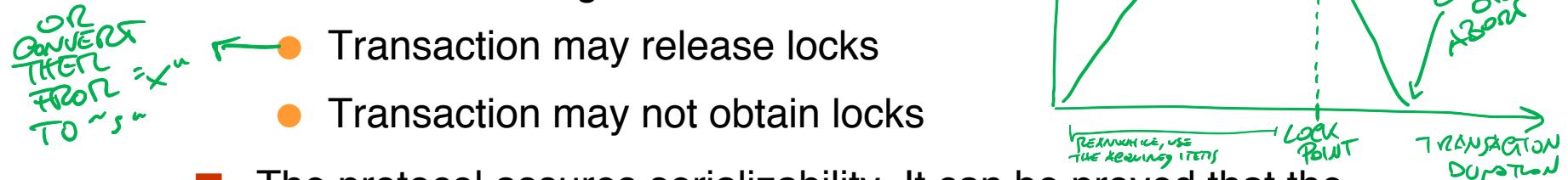
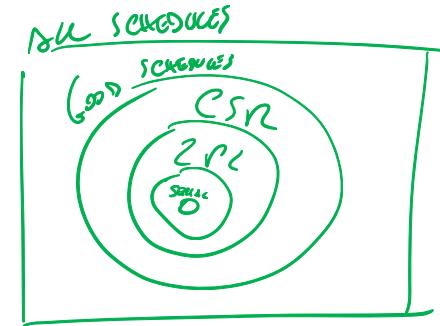
- Phase 1: Growing Phase

- Transaction may obtain locks
- Transaction may not release locks

- Phase 2: Shrinking Phase

- Transaction may release locks
- Transaction may not obtain locks

- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).



ACTUAL SYSTEMS
USE ZPL

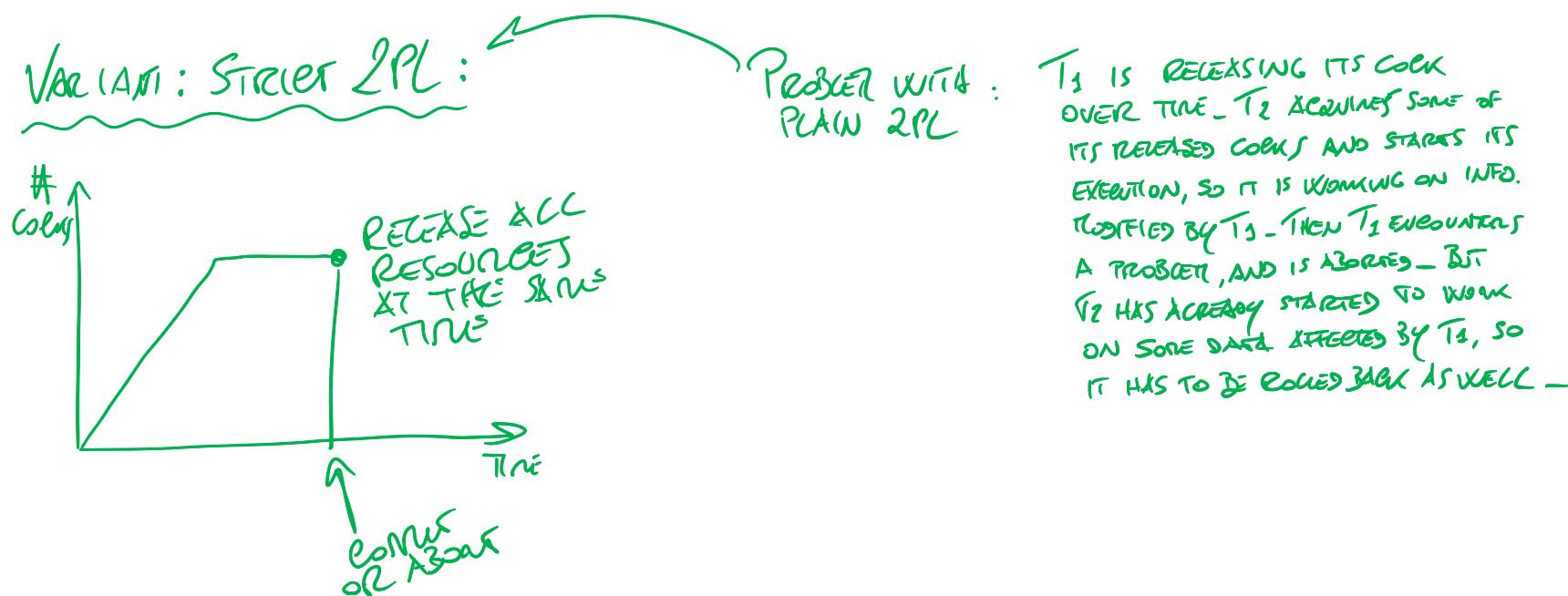
→ EFFICIENCY
WE NEED AN "ENQUE" CRITERION TO DECIDE WHETHER TO ACCEPT AN OPERATION REQUESTED BY A TRANSACTION (E.g. TWO WORK IN A POST-HOC WAY INSTEAD)

SO, YOU CANNOT
INTERCAVE
ACQUISITION AND
RELEASE OF
RESOURCES



The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.





Lock Conversions

- Two-phase locking with lock conversions:
 - First Phase:
 - can acquire a lock-S on item
 - can acquire a lock-X on item
 - can convert a lock-S to a lock-X (upgrade)
 - Second Phase:
 - can release a lock-S
 - can release a lock-X
 - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.



Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read(D)** is processed as:

```
if  $T_i$  has a lock on  $D$ 
  then
    read( $D$ )
  else begin
    if necessary wait until no other
      transaction has a lock-X on  $D$ 
    grant  $T_i$  a lock-S on  $D$ ;
    read( $D$ )
  end
```



Automatic Acquisition of Locks (Cont.)

- **write(D)** is processed as:

if T_i has a **lock-X** on D

then

write(D)

else begin

if necessary wait until no other transaction has any lock on D ,

if T_i has a **lock-S** on D

then

upgrade lock on D to **lock-X**

else

grant T_i a **lock-X** on D

write(D)

end;

Does somebody of you
spot a problem with this
policy of waiting/granting/releasing locks?

- All locks are released after commit or abort

(If we consider strict 2-P L)



NOW CONSIDER
THIS SITUATION

Deadlocks

- Consider the partial schedule

T_3	T_4
lock-x (B) read (B) $B := B - 50$ write (B) <i>lock-x (A)</i>	lock-s (A) read (A) lock-s (B) <i>wait</i> ↘ ↗ <i>wait</i>

- Neither T_3 nor T_4 can make progress — executing **lock-S(B)** causes T_4 to wait for T_3 to release its lock on B , while executing **lock-X(A)** causes T_3 to wait for T_4 to release its lock on A .
- Such a situation is called a **deadlock**.
 - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.



Deadlocks (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks.
- In addition to deadlocks, there is a possibility of **starvation**.
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
 - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
 - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

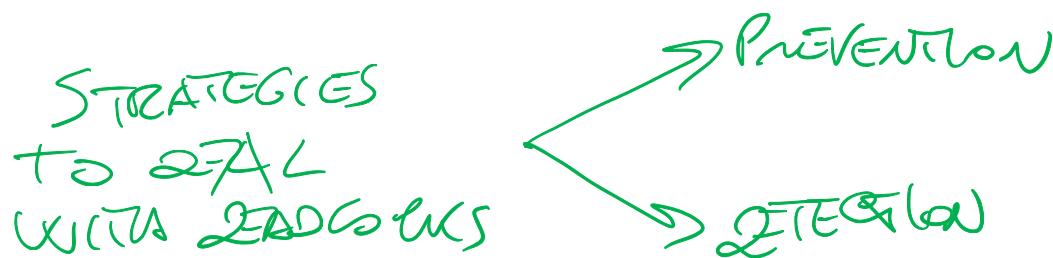
↓
AGEING

FOR INSTANCE, IT MIGHT
PREFER TO GRANT
SHARED LOCKS SINCE
THEY ARE LESS "COSTLY"



Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked





Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- **Deadlock prevention** protocols ensure that the system will never enter into a deadlock state. Some prevention strategies :
 - Require that each transaction locks all its data items before it begins execution (predeclaration). *→ TOO RESTRICTIVE, RAY NEVER FIND ALL OF THEM AVAILABLE*
 - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.

THEN
WE
BREAK
CYCLES

A TRANSACTION RAY
PREFER IS DIFFERENT
ORDER . . .



More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
 - older transaction may wait for younger one to release data item.
(older means smaller timestamp)
Younger transactions never wait for older ones; they are rolled back instead.
 - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
 - older transaction *wounds* (forces rollback ~~of~~) younger transaction instead of waiting for it. Younger transactions may wait for older ones.

IF T_i WAITS ON A RESOURCE HELD BY T_j , WHICH IS YOUNGER THAN T_i , THEN T_i KILLS T_j —

KILLING A TRANSACTION IS NOT A TRAGEDY, SINCE ALL OPERATIONS IT HAS DONE UP TO THAT POINT ARE UNDONE, AND THE ENTIRE TRANSACTION IS THEN RESTARTED —



Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Timeout-Based Schemes:

- a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
- Thus, deadlocks are not possible
- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

→ *ROST USED ALONG DEADLOCK PREVENTION STRATEGIES - KIND OF SUICIDE BY THE TRANSACTION -*

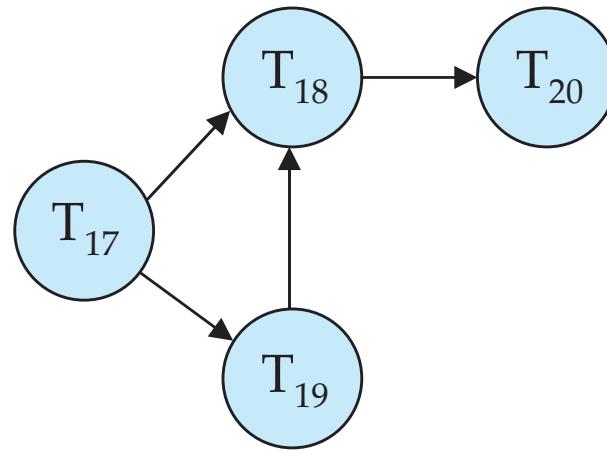


Deadlock Detection

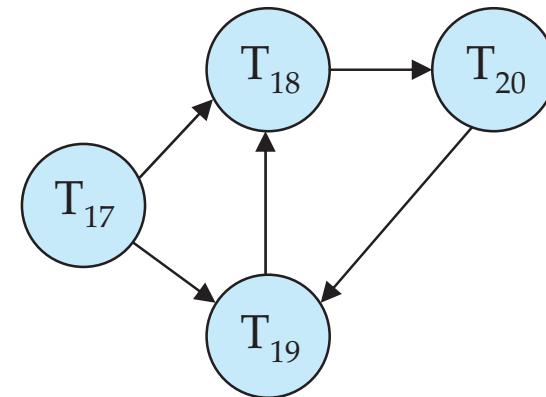
- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V, E)$,
 - V is a set of vertices (all the transactions in the system)
 - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E , then there is a directed edge from T_i to T_j , implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i .
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

ROST USED
SOLUTIONS

→ Thread-based deadlock prevention
→ Deadlock detection with graphs