



4. Database Management Systems (4/4)

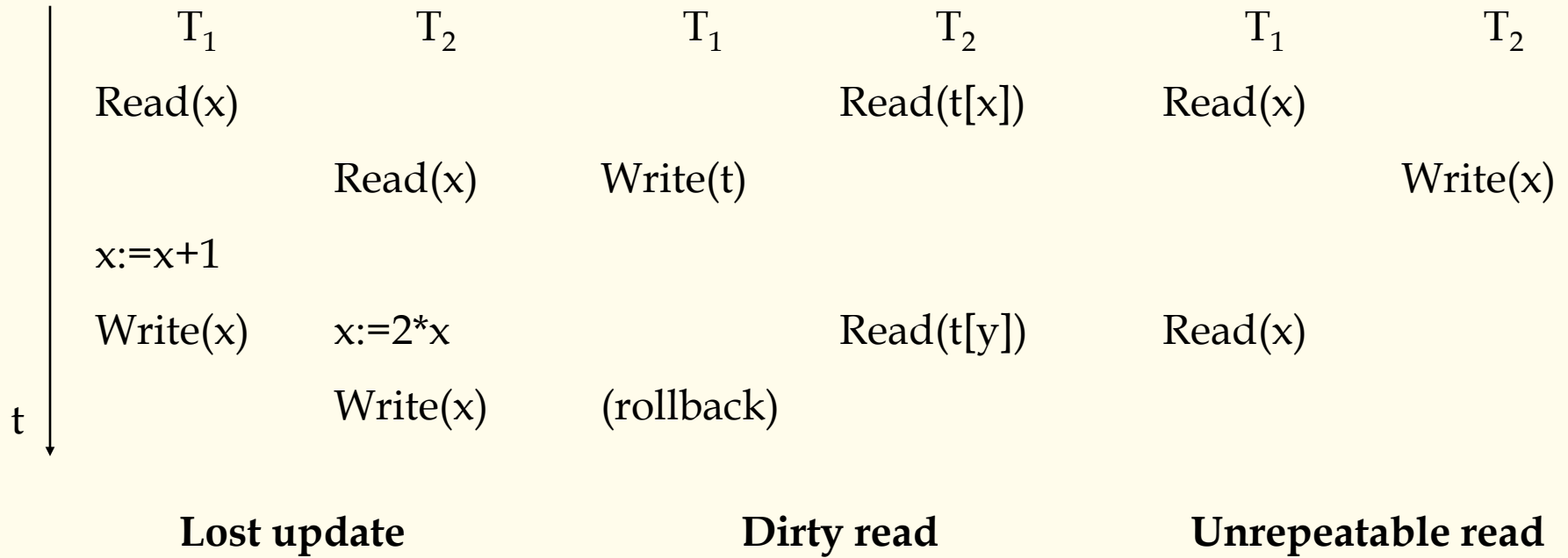


4.6 Concurrency Control

4.6.1 Introduction

In multi users DBMS, permit multi transaction access the database concurrently.

- Why concurrency?
 - 1) Improving system utilization & response time.
 - 2) Different transaction may access to different parts of database.
- Problems arise from concurrent executions



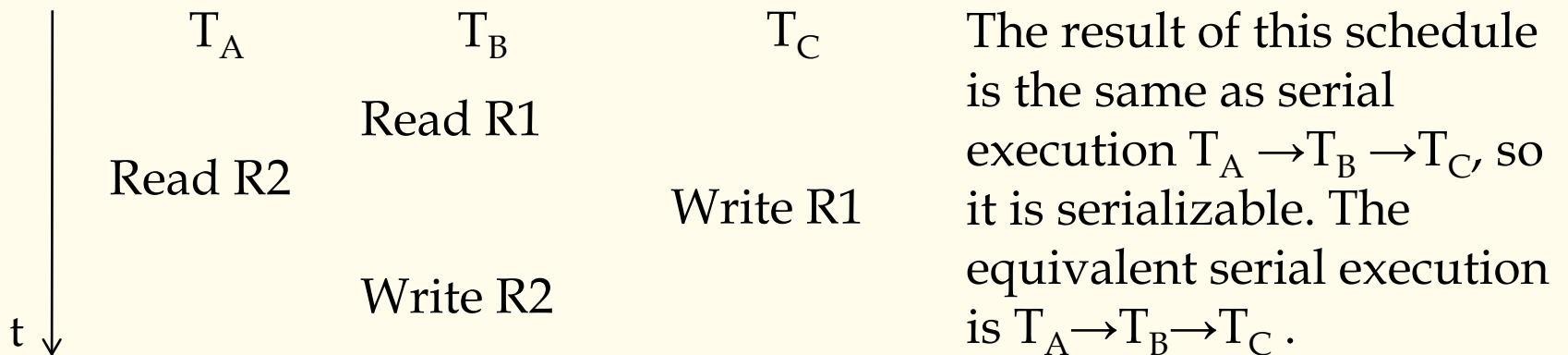
So there may be three kinds of conflict when transactions execute concurrently. They are write – write, write – read, and read – write conflicts. Write – write conflict must be avoided anytime. Write – read and read – write conflicts should be avoided generally, but they are endurable in some applications.



4.6.2 Serialization --- the criterion for concurrency consistency

Definition: suppose $\{T_1, T_2, \dots, T_n\}$ is a set of transactions executing concurrently. If a schedule of $\{T_1, T_2, \dots, T_n\}$ produces the same effect on database as some serial execution of this set of transactions, then the schedule is serializable.

Problem: different schedule \rightarrow different equivalent serial execution \rightarrow different result? (yes, $n!$)



4.6.3 Locking Protocol

Locking method is the most basic concurrency control method. There maybe many kinds of locking protocols.

(1) X locks

Only one type of lock, for both read and write.

Compatibility matrix : NL—no lock X—X lock
 Y —compatible N—incompatible

	NL	X
NL	Y	Y
X	Y	N

T_A
 X_lock R
 Update R
 ⋮
 X_unlock R
 EOT

T_B
 X_lock R
 wait
 ↓
 X_lock R
 Read R
 ⋮



*Two Phase Locking

- *Definiton1:* In a transaction, if all locks precede all unlocks, then the transaction is called two phase transaction. This restriction is called two phase locking protocol.
- *Definition2:* In a transaction, if it first acquires a lock on the object before operating on it, it is called well-formed.

- *Theorem:* If S is any schedule of well-formed and two phase transactions, then S is serializable. (proving is on p151)

	T_1	T_2
Growing phase	Lock A Lock B Lock C ⋮	Lock A Lock B Unlock A Unlock B
Shrinking phase	Unlock A Unlock B Unlock C	Lock C ⋮ Unlock C
	2PL	not 2PL



Conclusions :

- 1) Well-formed + 2PL : serializable
- 2) Well-formed + 2PL + unlock update at EOT: serializable and recoverable. (without domino phenomena)
- 3) Well-formed + 2PL + holding all locks to EOT: strict two phase locking transaction.

(2) (S,X) locks

S lock --- if read access is intended.

X lock --- if update access is intended.

	NL	S	X
NL	Y	Y	Y
S	Y	Y	N
X	Y	N	N

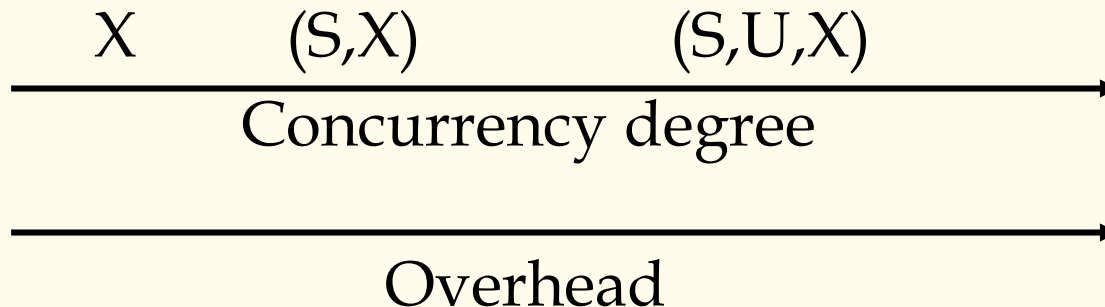


(3) (S,U,X) locks

U lock --- update lock. For an update access the transaction first acquires a U-lock and then promote it to X-lock.

Purpose: shorten the time of exclusion, so as to boost concurrency degree, and reduce deadlock.

	NL	S	U	X
NL	Y	Y	Y	Y
S	Y	Y	Y	N
U	Y	Y	N	N
X	Y	N	N	N





4.6.4 Deadlock & Live Lock

Dead lock: wait in cycle, no transaction can obtain all of resources needed to complete.

Live lock: although other transactions release their resource in limited time, some transaction can not get the resources needed for a very long time.

T_A
X_lock R1

⋮

X_lock R2

wait

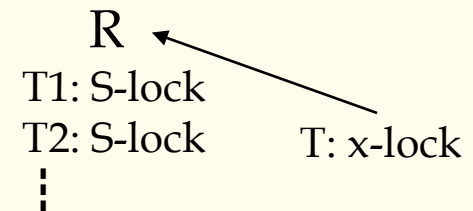


T_B
X_lock R2

⋮

X_lock R1

wait



- Live lock is simpler, only need to adjust schedule strategy, such as FIFO
- Deadlock: (1) Prevention(don't let it occur); (2) Solving(permit it occurs, but can solve it)



(1) Deadlock Detection


- 1) Timeout: If a transaction waits for some specified time then deadlock is **assumed** and the transaction should be aborted.
- 2) Detect deadlock by wait-for graph $G = \langle V, E \rangle$
 V : set of transactions $\{T_i \mid T_i \text{ is a transaction in DBS } (i=1,2,\dots,n)\}$
 E : $\{ \langle T_i, T_j \rangle \mid T_i \text{ waits for } T_j (i \neq j) \}$
 - If there is cycle in the graph, the deadlock occurs.
 - When to detect?
 - 1) whenever one transaction waits.
 - 2) periodically



- What to do when detected?
 - 1) Pick a victim (youngest, minimum abort cost, ...)
 - 2) Abort the victim and release its locks and resources
 - 3) Grant a waiter
 - 4) Restart the victim (automatically or manually)

(2) Deadlock avoidance

- 1) Requesting all locks at initial time of transaction.
- 2) Requesting locks in a specified order of resource.
- 3) Abort once conflicted.
- 4) Transaction Retry



Every transaction is uniquely time stamped. If T_A requires a lock on a data object that is already locked by T_B , one of the following methods is used:

- a) Wait-die: T_A waits if it is older than T_B , otherwise it “dies”, i.e. it is aborted and automatically retried with original timestamp.
- b) Wound-wait: T_A waits if it is younger than T_B , otherwise it “wound” T_B , i.e. T_B is aborted and automatically retried with original timestamp.

In above, both have only one direction wait, either older \rightarrow younger or younger \rightarrow older. It is impossible to occur wait in cycle, so the dead lock is avoided.