

# 15 Concurrency control

## 15.1 Serializability and Concurrency Control

### 15.2 Locking

- 15.2.1 Lock protocols
- 15.2.2 Two phase locking
- 15.2.3 Strict transactional protocols
- 15.2.4 Lock conflicts and Deadlocks
- 15.2.5 Lock modes
- 15.2.6 Deadlock detection, resolution, avoidance

### 15.3 Nonlocking concurrency control

- 15.3.1 Multiversion cc
- 15.3.2 Optimistic cc: forward / backward oriented
- 15.3.3 Time stamp ordering

### 15.4 Synchronizing index structures

### 15.4 Distributed transactions: Two Phase Commit (short)

Lit.: Eickler/ Kemper chap 11.6-11.13, Elmasri /Navathe chap. 20, Garcia-Molina, Ullman, Widom: chap. 18

# Concurrency control

- No explicit locking in application programs
  - error prone,
  - responsibility of scheduler (and lock manager)

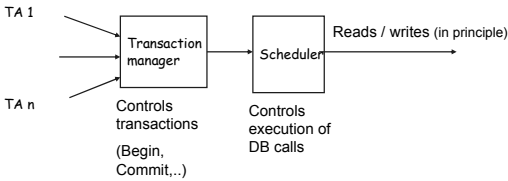
In most DBS also explicit locking allowed in addition to implicit locking by scheduler. Use with care!

- Not considered here: **transaction independent locking**, e.g. writing a page p to disk requires a short term lock on p

# Concurrency control ...and serializability

## Wanted:

effective **real-time scheduling** of operations with guaranteed serializability of the resulting execution sequence.



# Optimistic vs. pessimistic

- Locking is **pessimistic**
  - Assumption: during operation op [x] of TA1 a (potentially) conflicting operation op'[x] of TA2 will access the same object x
  - This has to be avoided by locking x before accessing x

S: r1[y], r3[u], r2[y], w1[y], w2[x], w1[x], w2[z], c2, w3[x]

TA1 read x, must wait until TA2 has committed

Note: call sequence and execution sequence different

- An **optimistic** strategy would be:
  - Perform all operations on a copy of the data. Check at the end - before commit - if there were any conflicts.
  - If no: commit, else abort (rollback) - more or less

# Concurrency control

## Concurrency control in DBS

- methods for scheduling the operations of database transactions in a way which guarantees serializability of all transactions ("between system start and shutdown")
- Primary concurrency control methods
  - Locking (most important)
  - Optimistic concurrency control
  - Time stamps
  - Multiversion CC

# 15.2.1 Lock protocols

## Simple object locking

Lock each object before writing / writing, unlock when operation finished

⇒ schedule will not be serializable (why?)

### Example

l1(x) r1(x) ul1(x) l2(x) w2(x) ul2(x) l1(x) w1(x) ul(x)

⇒ lost update ⇒ useless

## Lock protocols

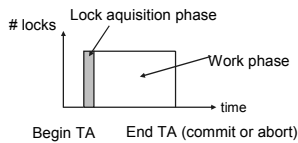
### • Preclaiming

- Acquire **all locks** needed before performing an operation
- release, if you do not get all of them. Try again. *race condition, transaction could starve!*
- Execute transaction
- Release locks

Preclaiming serializable?  
Why (not) ?

Bad: objects to be processed may not be known in advance.

Not used in DBS.



HS / DBS05-19-CC 7

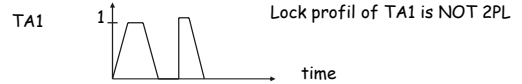
## Concurrency control 2PL

### • Why no lock after unlock?

#### • Example:

```
lock1(x), r1[x], x:=x*10, ulock1(x)
lock2(x), r1[x], x:=x+1, w2[x] ulock2(x)
lock1(x), w1(x), ulock1(x)
```

results in a lost update



⇒ Rule 3 is essential

HS / DBS05-19-CC 10

## 12.3 Two phase locking (2PL)

### The 2PL protocol

1. Each object referenced by  $TA_i$  has to be locked before usage
2. Existing locks of other TA's will be respected
3. No lock is requested by a  $TA_i$ , if a lock has been released by the same transaction  $TA_i$   
("no lock after unlock")
4. Locks are released at least at commit time
5. A requests of a lock by a TA which it already holds, has no effect.

...

HS / DBS05-19-CC 8

## Concurrency control 2PL

### 2-Phase locking theorem

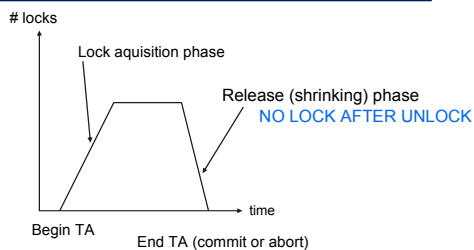
If all transactions follow the 2-phase locking protocol, the resulting schedule is serializable

#### • Proof sketch:

- Suppose a resulting schedule is not serializable. when using 2PL ⇒ conflict graph contains a cycle ⇒ there are transactions  $TA_1$  and  $TA_2$  with conflict pairs  $(p, q)$  and  $(q', p')$ ,  $p, p'$  atomic operations of  $TA_1$ ,  $q, q'$  of  $TA_2$ ,  $p, q$  access the same object  $x$ , and  $q', p'$  an object  $y$  (assuming a cycle of length 1, induction for the general case)

→ HS / DBS05-19-CC 11

## Concurrency control 2PL



- Locked objects may be read / written already in lock acquisition phase

HS / DBS05-19-CC 9

## Concurrency control 2PL

Let e.g.  $(p, q) = (r1[x], w2[x]),$   
 $(q', p') = (w2[y], w1[y])$

Analyze all of the possible execution sequences:

$p, q, q', p$   
 $p, q', q, p'$   
 $q', p, q, p'$   
 $q', p, p', q$   
 $q', p', p, q$

T2: Lock y, T1: Lock x, T2: Lock x, T1: Lock y

T1 must have released lock on x and acquired one on y (or T2 must have acquired after release)  
Violates 2-phase rule! Contradiction to assumption that all follow TAs use 2PL protocol

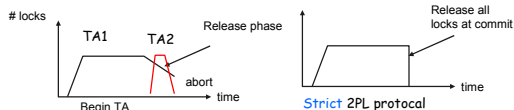
Same holds for the other possible sequences ⇒ Theorem

Note: serializability does not imply 2PL, i.e. there are serializable schedules which do not result from a 2PL scheduler

HS / DBS05-19-CC 12

### 15.2.3 Strict concurrency protocols

Locking protocol is **strict** if locks are released at commit / abort.



- A different transaction TA2 could have used an object x which was unlocked by TA1 in the release phase
  - no problem, if TA1 commits
  - if TA1 aborts, TA2 has used a wrong state of x TA2 has to be aborted by the system
- May happen recursively: cascading abort, **bad ...**
- Strict 2PL: Release** all locks at commit point HS / DBS05-19-CC 13

### Lock modes

#### Lock modes and lock compatibility

RX – model: read (R) and eXclusive(X) locks (or: write locks)

	holder	
	R	X
requester	R +	-
	X -	-

R-lock same as Shared (S) lock

Lock compatibility matrix

- Lock compatibility** in the RX model:
  - Objects locked in R-mode may be locked in R-mode by other transactions(+)
  - Objects locked in X-mode may not be locked by any other transaction in any mode. Lock conflict: requesting TA has to wait

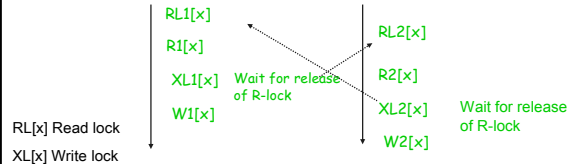
### 15.2.4 Lock conflicts and deadlocks

- Lock conflict**
  - Two or more processes request an exclusive lock for the same object
- Deadlock**
  - Locking: threat of deadlock
    - No preemption
    - No lock release in case of lock conflicts
  - ⇒ Two-Phase locking may cause deadlocks
  - $L_i[x]$  = Transaction i requests lock on x
  - $U_i[x]$  = Transaction i releases lock on x
  - Lock sequence:  $L_1[x], L_2[y], \dots, L_i[y], L_2[x]$  causes deadlock
  - How to deal with deadlocks? → see below

### Reduce deadlock threat

Deadlocks caused by typical read / write sequences

TA1: read account\_record x; incr(x.balance); write account\_record  
 TA2: read account\_record x; incr(x.balance); write account\_record



- ⇒ **Read-Update-Exclusive Model (RUX)**

### 15.2.5 Lock modes

- Primary goal**
  - no harmful effects (lost update, ...)
- Secondary goal**
  - Degree of parallelism should be as high as possible, even when locking is used
  - Low deadlock probability, if any
- Ways to increase parallelism**
  - Compatible locks (read versus write semantics)
  - Different lock granularity
  - Application semantics
  - No locks, optimistic cc

### Lock modes: RUX

#### RUX Lock protocol

- Transactions which read and subsequently update an object y request a U-lock, upgrade to X-lock before write
- Read locks cannot be upgraded
- U-locks incompatible with U-locks ⇒ deadlock-thread avoided
- U / R-lock compatibility asymmetric, why?

	holder			
	R	U	X	
requester	R +	-	-	
	U +	-	-	
	X -	-	-	

How does DBS know, that update is intended?

## Lock modes

### Hierarchical locking

- One single **lock granularity** (e.g. records) insufficient, large overhead when many rows have to be locked
- Most DBS have at least two lock granularities: **row locks** and **table locks**

Issue:  $TA_i$  wants to lock table R

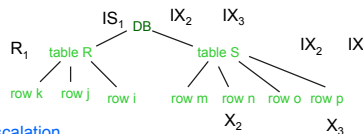
- some rows of R locked by different transactions
- ⇒ different lock conflict as before:  $TA_i$  is **waiting for release of all record locks**
- No other TA should be able to lock a record, otherwise  $TA_i$  could starve

HS / DBS05-19-CC 19

## Concurrency control Lock modes

### Hierarchical locking

- An object O on level i contains all objects x on level i+1
- Locks of O lock all subordinate objects x
- If a subordinate object x (level i+1) is locked, this is indicated by an intention lock on level i



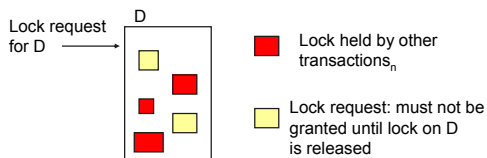
#### Lock escalation

If too many objects x on level i+1 are locked by a transaction, it may be converted into one lock on level i

HS / DBS05-19-CC 22

## Concurrency control

### Locks of different granularity



Efficient implementation of this type of situation??

HS / DBS05-19-CC 20

## Lock modes

### Hierarchical locking (cont)

- Advantage: **one lookup** is sufficient to check if a lock on higher level (say on a table) can be granted
- **Protocol**: if a TA wants to lock an object on level i in mode <M> (X or R), lock all objects on higher level (on the path to root) in I<M> – mode
- Easy to check, if the locks on all subordinate objects are released: implement I<M>-lock as a counter

requester	holder			
	IR	IX	R	X
IR	+	+	+	-
IX	+	+	-	-
R	+	-	+	-
X	-	-	-	-

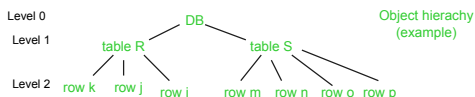
Compatibility matrix

How to combine with U-lock mode?

HS / DBS05-19-CC 23

## Lock modes: Hierarchical locking

### Intention locks



- Feature of **intention locks** for hierarchical locking: for each lock mode, there is an intention lock, e.g. for RX-lock modes: IR and IX
- Semantics: A TA holds a IM-lock on an object D on level i, if and only if it holds an M-lock on an object D' on level j > i subordinate to D

HS / DBS05-19-CC 21

## 15.2.6 Deadlock detection, resolution, avoidance

### Deadlocks

... can happen with 2PL protocol (see above)

- Release of a lock could break rule 4

$XL1[x]$ ,  $XL2[y]$ ,  $XL1[y]$  →  $TA1$ : WAIT for  $XU2[y]$ ,  $XL2[x]$  →  $TA2$ : WAIT for  $XU1[x]$

- Note: **deadlocks very different from lock conflicts**:

...  $XL1[x]$ ,  $XL2[y]$ ,  $XL1[y]$  →  $TA1$ : WAIT for  $XU2[y]$ ,  $XL2[z]$ ,  $w2[y]$ ,  $w2[z]$ ,  $XU2[y]$ , ...

Lock conflict, y is locked by TA2, TA1 waits for unlock

Lock conflict resolved by  $XUlock2[x]$ , TA1 proceeds

Not schedules, but call sequences including lock / unlock operations by the scheduler

HS / DBS05-19-CC 24

## Deadlock

### Detection and resolving deadlocks

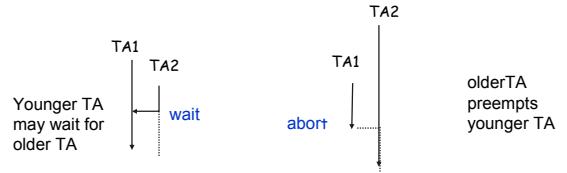
- Cycle check in Wait-for-graph
  - Waiting of TA1 for release of lock on x by TA2 is indicated by an arc from TA1 to TA2 labeled "x"
  - Cycles indicate deadlock
  - In a distributed environment, deadlocks may involve different systems. How to detect cycles?
  - One of the waiting transaction ("victim") is rolled back
  - Which one??
- Timeout
  - If TA has been waiting longer than the time limit, it is aborted.
  - Efficient but may roll back innocent victims (deadlock does not exist)

Oracle: WF-graph in central DB, timeout in distributed

HS / DBS05-19-CC 25

## Deadlock avoidance

### Wound / Wait



If  $TS(TA1) < TS(TA2)$  then TA2.wait else TA1.abort

HS / DBS05-19-CC 28

## Deadlock avoidance

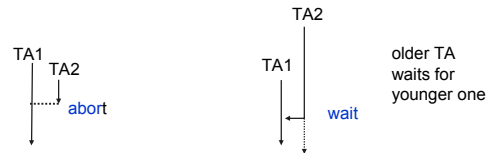
### Avoiding deadlocks

- Deadlocks only occur, if **no preemption**
- Force preemption by the lock manager
- TA t is preempted  $\Rightarrow$  forced to rollback
- Preemption  $\Rightarrow$  no deadlocks, but living transactions may be killed
- **Wait/Die - Wound/Wait** : Basic idea
  - Solve lock conflicts by rollback of one of the conflicting transactions....
  - .... but not always
  - **Rollback dependent** on the relative **age** of the transactions
  - Time stamp for each transaction

HS / DBS05-19-CC 26

## Deadlock avoidance

### Wait / Die



If  $TS(TA1) < TS(TA2)$  then TA2.abort else TA2.wait

No deadlocks! Why?

Aborted transaction restarts with old timestamp in order to avoid starvation

HS / DBS05-19-CC 29

## Deadlock avoidance

- **Wound/Wait – Wait / Die** methods
  - Each transaction  $TA_i$  has an **initial timestamp**  $TS(TA_i)$
  - If  $TA_2$  requests a lock on x and there is a lock conflict with  $TA_1$ , one of them **may be aborted**

TA2 requests lock which TA1 holds:

### - WOUND / WAIT

if  $ts(TA1) < ts(TA2)$  then TA2.WAIT else TA1.ABORT  
 Abort lock holding TA if younger than requesting, else wait

### - WAIT / DIE

if  $ts(TA1) < ts(TA2)$  then TA2.ABORT else TA2.WAIT  
 Abort requesting TA if younger, else wait

HS / DBS05-19-CC 27

## 15.3 Non-locking protocols (more or less)

### 15.3.1 Multiversion CC:

$r1[x] w1[x] r2[x] w2[y] r1[y] w1[z] c1 w2[a] c2$   
 not serializable.

If  $r1[y]$  had arrived at the scheduler before  $w2[y]$  the schedule would have been serializable.

- Main idea of **multiversion concurrency control** : Reads should see a consistent (and committed) state, which might be older than the current object state.
- Necessary: **Different version** of an object
- Read and write locks compatible (!)
- In the example: TA2 must not write its version of y before TA1 has released lock on y
- Particular important in practice: 2 versions

Arrows from TA2-ops to conflicting TA1-ops

HS / DBS05-19-CC 30

## Multiversion concurrency

### Lock based MVCC ("MV2PL")

- Read locks always granted, write lock if object not write locked =>
- **two versions**: consistent one and writable private copy
- When TA wants to write modified copy of x into DB it has to wait until all readers of x have released read lock
- write is delayed to ensure consistent read using a certify lock

	R	W	C
R	+	+	-
W	+	-	-
C	-	-	-

C = Certify

HS / DBS05-19-CC 31

## MVCC / Read Only TAs: Example

call sequence: TA1, TA4 and TA5 are RO

R1(x) r2(x)w2(x)r3(x)r2(y)R4(z)w2(y)c2R4(x)c4w3(x)R5(z)c3R1(y)c1R5(x)c5

R1(x0)\_\_\_\_\_R1(y0)c1

r2(x0)w2(x0)\_r2(y0)\_\_\_\_\_w2(y)c2

r3(x)..... blocked.....r3(x2)\_w3(x3)c3

R4(z0)\_\_\_\_\_R4(x0)c4

R5(z0)\_\_\_\_\_R5(x2)c5

R1(y0): there exists a newer version y2, but RO\_TA1 is older  
 R5(x2): reads x2 since TA3 which produces x3, commits after TA 5 begins  
 R4(x0): same with TA2, which produces x2  
 TA3 has been blocked, since TA2 holds lock on x, r3(x2) after TA2 committed

HS / DBS05-19-CC 34

## Multiversion concurrency

### Two-version-2PL MVCC

- has only **one uncommitted** version, one consistent ("current") version because writes are incompatible
- Readers benefit, not writers
- may be generalized to more than one uncommitted
- is most important in practice

Scheduler:

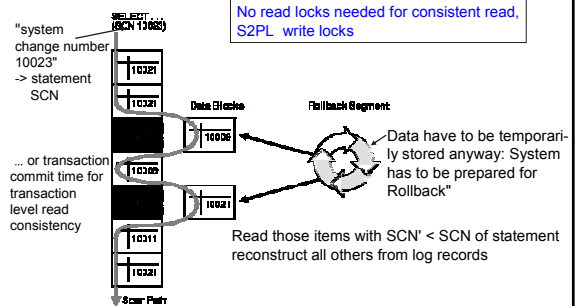
- rl(x) : set read lock immediately on consistent version of x
- wl(x) : if not write locked, set write lock on x to produce a new uncommitted version
- cl(x) : if neither read-locked nor write-locked cl(x) is granted and x will be written as the new consistent version by the TA

Correctness? Deadlocks? Read locks needed?

HS / DBS05-19-CC 32

## MVCC: How to implement versions

### Read Only Multiple version CC (used in Oracle)



HS / DBS05-19-CC 35

## MVCC for Read only Transactions

- Read-only transactions always read the last consistent state
- Last consistent state for Read only TA R: last committed value(s) before R starts
- Idea:
  - Each TA has a timestamp ("begin TA")
  - **Update transactions** with TS t makes a new version of updated data x,y,.. at commit, version of x,y,.. is t
  - Read TA with timestamp t' read only those values the **version t of which is less than t'**
- Update TA use conventional 2PL protocol with S and X locks

HS / DBS05-19-CC 33

## 15 Concurrency control

### 15.1 Serializability and Concurrency Control

### 15.2 Locking

- 15.2.1 Lock protocols
- 15.2.2 Two phase locking
- 15.2.3 Strict transactional protocols
- 15.2.4 Lock conflicts and Deadlocks
- 15.2.5 Lock modes
- 15.2.6 Deadlock detection, resolution, avoidance

### 15.3 Nonlocking concurrency control

- 15.3.1 Multiversion cc
- 15.3.2 Optimistic cc: forward / backward oriented
- 15.3.3 Time stamp ordering

### 15.4 Synchronizing index structures

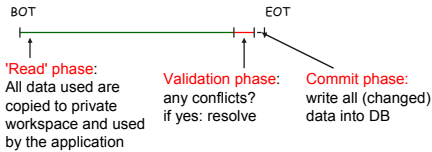
### 15.4 Distributed transactions: Two Phase Commit (short)

Lit.: Eickler/ Kemper chap 11.6-11.13, Elmasri / Navathe chap. 20, Garcia-Molina, Ullman, Widom: chap. 18

## Optimistic CC

### 15.3.2 Optimistic concurrency control

- Locks are expensive
- Few conflicts  $\Rightarrow$  retrospective check for conflicts cheaper
- Basis idea: all transactions **work on copies**, check for conflicts before write into DB
- if conflict: abort else commit



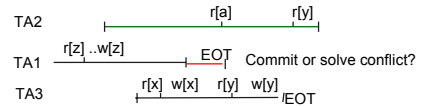
HS / DBS05-19-CC 37

## Optimistic CC: FOCC

### Forward oriented optimistic Concurrency control (FOCC)

- Forward looking validation phase:

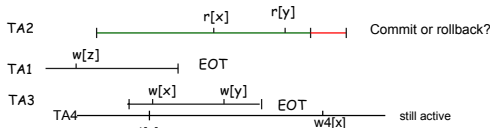
if there is a running transaction  $T'$  which read data written by the validating transaction  $T$  then solve the conflict (e.g. kill  $T'$ ), else commit



HS / DBS05-19-CC 40

## Optimistic CC: BOCC

### Backward oriented concurrency control (BOCC)



- **ReadSet**  $R(T)$  = data, transaction  $T$  has read in read phase
- **WriteSet**  $W(T)$  = data (copies!),  $T$  has changed in read phase

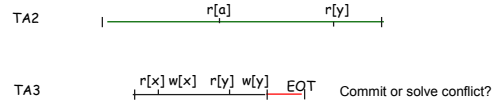
Assumption:  $W(T) \subseteq R(T)$  - necessary?

Example above:  $x, y \in R(T2)$ ,  $x, y \in W(T3)$ ,  $z \in W(T1)$

Conflict? Let  $x \in R(T)$ .  $T$  wants to validate.  
 If a different TA read  $x$ , but did not commit  $\Rightarrow$  no problem  
 If a different TA committed after  $BOT(T)$ : DB state of  $x$  may be different from  $x$  at  $BOT(T)$   $\Rightarrow$  conflict

HS / DBS05-19-CC 38

## Concurrency: Optimistic CC



FOCC\_validate( $T$ ): if for all running transactions ( $T'$ )  
 $R(T') \cap W(T) = \emptyset$   
 then  $T$ .commit // successful validation  
 else solve\_conflict( $T, T'$ )

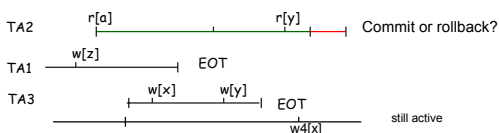
$R(T)$ : Read set of  $T$  at validation time of  $T$  (current read set)

HS / DBS05-19-CC 41

## Optimistic CC: BOCC

BOCC\_validate( $T$ ):

if for all transactions  $T'$  which committed after  $BOT(T)$ :  
 $R(T) \cap W(T') = \emptyset$  then  $T$ .commit // successful validation  
 else  $T$ .abort



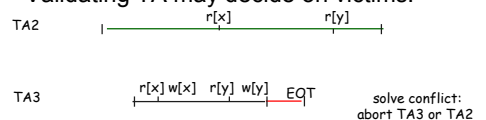
Shown: when they are needed! Consequence: More aborts than necessary:  
 $R(TA2) \cap W(TA3) \neq \emptyset$ . No abort when locking, not even a lock conflict.

Validation: what happens, if more than one TA validates?

HS / DBS05-19-CC 39

## Concurrency control Optimistic CC

- Validation of read only transactions  $T$ : FOCC guarantees successful validation !
- FOCC has greater flexibility  
Validating TA may decide on victims!



- **Issues** for both approaches:  
fast validation – only one TA can validate at a time.  
Fast and atomic commit processing,
- Useful in situation with few expected conflicts

HS / DBS05-19-CC 42

## Implementation of Read / Write sets

- Possible implementation of Read / Write sets: attach to each object  $x$  timestamp  $ts(x)$  of last write.
- Validating TA (T) checks if  $ts(x)$  changed since BOT(T)
- Important detail: timestamp of data item on disk? Access many disk records to validate? Expensive!
- "Records on disk are older than or equal to the records in buffer"

HS / DBS05-19-CC 43

## 15.3. 3 Time stamp ordering

### Time stamp ordering

Basic idea:

- assign timestamp when transaction starts
- if  $ts(t_1) < ts(t_2) \dots < ts(t_n)$ , then scheduler has to produce history equivalent to  $t_1, t_2, t_3, t_4, \dots t_n$

Timestamp ordering rule:

If  $pi[x]$  and  $qj[x]$  are conflicting operations,  
then  $pi[x]$  is executed before  $qj[x]$  ( $pi[x] < qj[x]$ )  
iff  $ts(ti) < ts(tj)$

HS / DBS05-19-CC 46

## Optimistic CC & Locking

- Combining locks with optimism
  - Example: high traffic reservation system
  - typical TA: check "seats\_avail > 0 ?" //seats\_avail is hot spot
    - if yes, do this and that;
    - write seats\_avail-1
  - seats\_avail is a hot spot object
  - Not the state per se is important but the predicate "seats\_avail > 0 ?"
  - Optimism: if pred is true at BOT then it will be true with high probability at EOT
  - But if not: abort

HS / DBS05-19-CC 44

## Timestamp ordering

- TO concurrency control guarantees conflict-serializable schedules:

If not: cycle in conflict graph  
cycle of length 2:  $ts(t_1) < ts(t_2) \wedge ts(t_2) < ts(t_1)$   
#  
induction over length of cycle  $\Rightarrow$  #

$\Rightarrow$  No cycle in conflict graph ✓

HS / DBS05-19-CC 47

## Optimistic CC & Locking

- Additional operations Verify and Modify:  
Verify P: check predicate P ("seats\_avail > 0?")  
//like read phase  
put "seats\_avail-1" into to\_do list  
rest of TA  
EOT:  
Modify: for all operations on to\_do list  
{ lock; verify once more;  
if 'false' rollback else write updates;}  
unlock all;
- Short locks, more parallelism
- If only decrement / increment operations: concurrent writing possible without producing inconsistencies
- Enhancement: Escrow locks – system guarantees that predicate still holds. Only ordered sets and inc / dec operations

HS / DBS05-19-CC 45

## TO Scheduler

- Basic principle:  
Abort transaction if its operation is "too late"  
Remember timestamp of last write of  $x$ :  $maxW[x]$   
and last read  $maxR[x]$

Transaction  $i$ :  $ti$  with timestamp  $ts(ti)$   
Operations:  $ri(x)$  /  $wi(x)$  -  $ti$  wants to read / write  $x$   
Scheduler state:  $maxR[x]$  /  $maxW[x]$   
timestamp of youngest TA  
which read  $x$  / has written  $x$

HS / DBS05-19-CC 48



## TO Scheduler: read

**Read:** TA  $t_i$  with timestamp  $ts(t_i)$  wants to read  $x : ri(x)$   
 $maxW[x] > ts(t_i)$ :  
 there is a younger TA which has written  $x$   
 $\Rightarrow$  contradicts timestamp ordering:  
 $t_i$  reads too late  
 $\Rightarrow$  abort TA  $t_i$ , restart  $t_i$

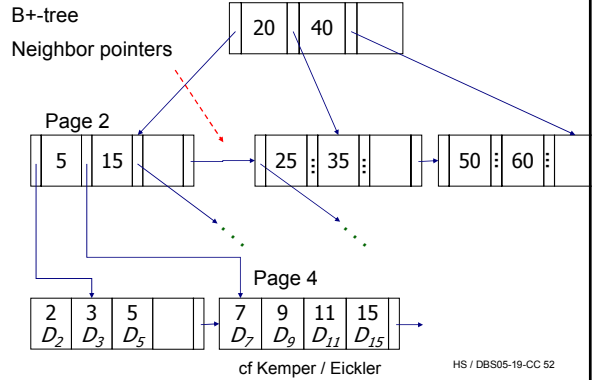
$maxW[x] < ts(t_i) \Rightarrow maxR[x] = ts(t_i)$ , go ahead

Example:  $-----|-----|----- >$   
 $wj(x) \quad ri(x) \quad ts(t_i) < ts(t_j)$

What would happen in a locking scheduler?

HS / DBS05-19-CC 49

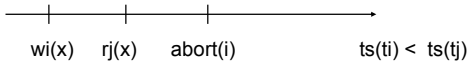
## 15.4. Synchronization of Indexstructures



## TO Scheduler: write

**Write:** TA  $t_i$  with timestamp  $ts(t_i)$  wants to write  $x : wi(x)$   
 $maxW[x] > ts(t_i) \vee maxR[x] > ts(t_i)$ :  
 $t_i$  but  $x$  has been *written or read by younger transaction*:  
 $\Rightarrow$  contradicts timestamp ordering  
 $\Rightarrow$  abort TA  $t_i$   
 otherwise:  $\Rightarrow$  schedule  $wi(x)$  for execution

Why abort ?



Dirty read! Solution: scheduler delays  $rj$  until TA committed  
 TA: the TA which was the last writer.

HS / DBS05-19-CC 50

## Synchronisation of index structures

### • Basic idea:

- Index structures are redundant, no reason to put them into transactional brackets
- Concurrent operation on B+-tree: short page locks Sufficient??

Scenario: TA1 searches for rec 15  
 has processed page 2 and found pointer to p4

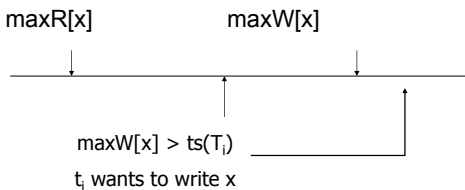
TA2 inserts rec 14: page split!  
 $\Rightarrow$  TA1 will not find rec 15

Solution: TA1 find rec in a right neighbor

HS / DBS05-19-CC 53

## Thomas Write Rule

- Idea: younger write overwrites older write without changing effect of timestamp ordering

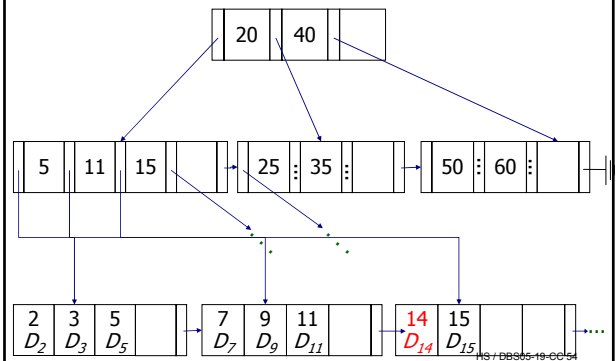


Rules for Writer T with timestamp  $ts(T)$ :

1.  $maxR[x] > ts(T)$  abort T
2.  $maxW[x] > ts(T)$  skip write // Thomas write rule
3. otherwise write  $x$ ,  $maxW[x] = TS(T)$

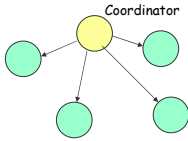
HS / DBS05-19-CC 51

## Synchronization of Indexstructures



## 15.5 Distributed Transactions Intermezzo

### • Configuration



- Different **resource managers** involved in the transaction  
e.g. database systems, mail server, file system, message queues,...
- **Asynchronous** and independent
- **One transaction coordinator** can be a resource manager or not
- One or more **participants**

Examples: Transfer of money/shares / ... from Bank A to B  
ECommerce systems  
All kinds of processing in decentralized systems

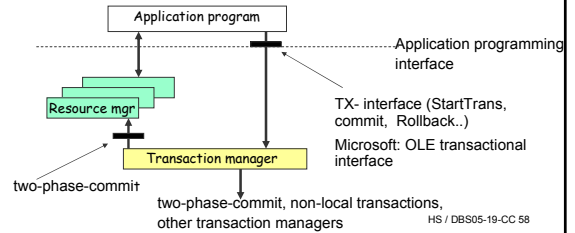
Frequently used in multi-tier architectures: middle tier accesses different databases.

HS / DBS05-19-CC 55

## Distributed Transactions

### Transaction managers: the X/Open transaction model

- Independent systems which coordinate transactions involving multiple resource managers as a service for application programs

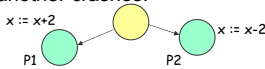


HS / DBS05-19-CC 58

## Distributed Transactions Intermezzo

### • Problems

- No problem ... if all systems work reliable
- Deadlock? Difficult to detect: use optimistic locking
- Obvious inconsistencies, if one participant commits, another crashes:



Coordinator: COMMIT  
P1 commits  
P2 crashes, undo??  
Introduces global inconsistency

### • Assumptions

- Each resource manager has a transactional recovery system (log operations, commit, rollback)
- There is exactly one commit coordinator, which issues commit for a transaction exactly once
- A transaction has stopped processing at each site before commit is issued

HS / DBS05-19-CC 56

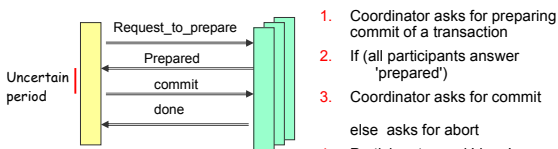
## Summary: Transactions and concurrency

- Transactions: Very important concept
- Model for consistent, isolated execution of TAs
- Scheduler has to decide on interleaving of operations
- Serializability: correctness criterion
  - 2-phase-locking
  - hierarchical locks
  - variation in order to avoid deadlocks :  
wound / wait, wait / die
  - optimistic concurrency control
  - multiversion cc

HS / DBS05-19-CC 59

## Distributed Transactions

### • The Two Phase Commit protocol (2PC)



1. Coordinator asks for preparing commit of a transaction
2. If (all participants answer 'prepared')
3. Coordinator asks for commit  
else asks for abort
4. Participants send 'done'

- After prepare phase: participants are ready to commit or to abort; they still hold locks
- If one of the participants does not reply or is not able to commit for some reason, the global transaction has to be aborted.
- Problem: if coordinator is unavailable after the prepare phase, resources may be locked for a long time

HS / DBS05-19-CC 57