

Chapter 6

Isolation: Synchronization and Scheduler

Agenda

- System architecture
- Basic locking schedulers
- Multiple granularity locking schedulers
- Tree locking schedulers
- Predicate locking
- Non-locking schedulers
- Optimistic schedulers

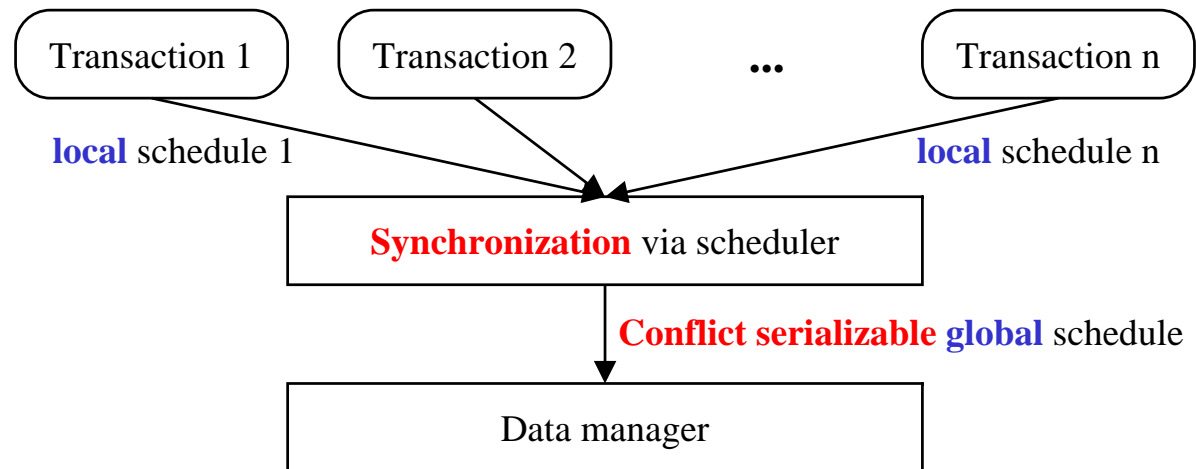


All CSR!

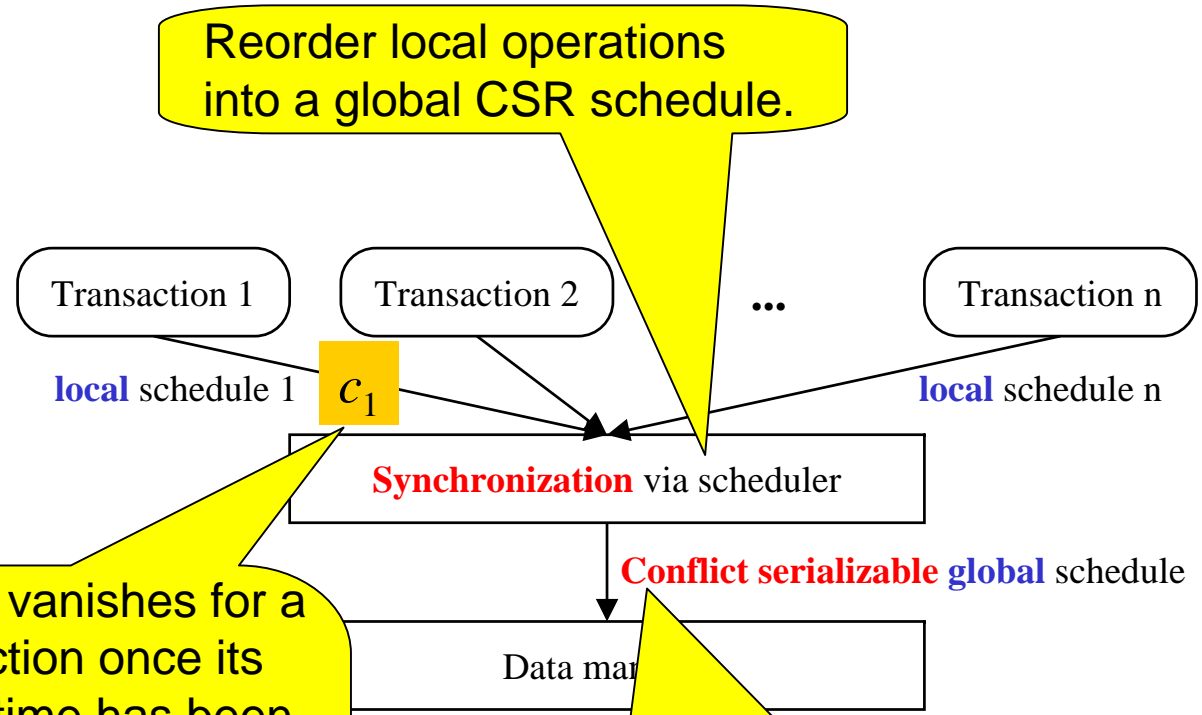
System architecture

Synchronization (1)

Simplified architecture to deal with CSR isolation



Synchronization (2)

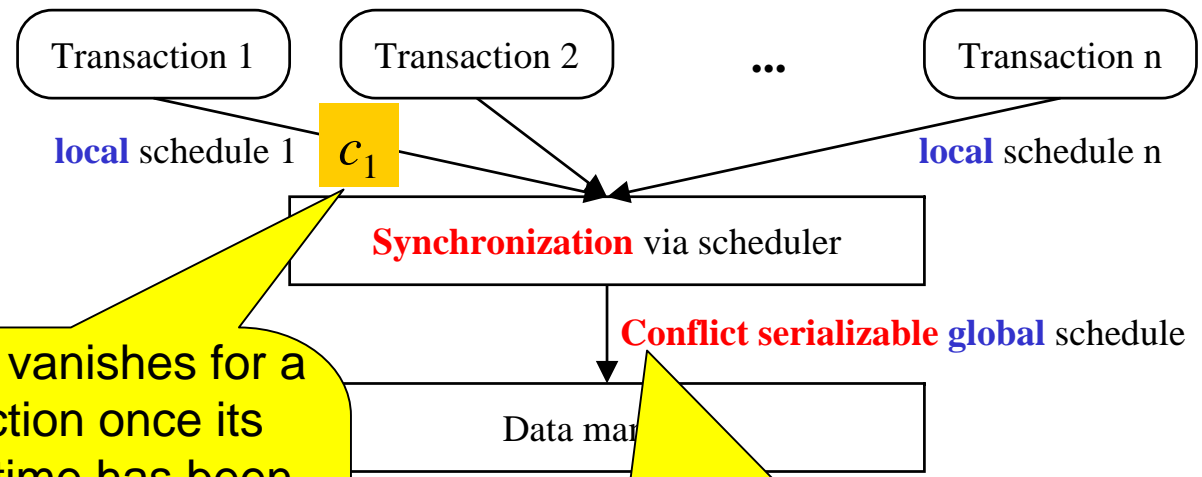


The freedom vanishes for a given transaction once its equivalence time has been reached. \Rightarrow At the latest when c_i has been issued. \Rightarrow Scheduler must have decided by then where to place the transaction in the serial schedule.

Remember: Given a set of transactions there may be several equivalent serial histories. \Rightarrow Scheduler has some freedom of choice. Prefix commit closure guarantees the freedom.

Synchronization (3)

Choice is limited because scheduler has no control over the order of arrival of operations and conflicts. \Rightarrow Schedule evolves as a mixture of arrival situation and optimizer strategy of the scheduler.



The freedom vanishes for a given transaction once its equivalence time has been reached. \Rightarrow At the latest when c_i has been issued. \Rightarrow Scheduler must have decided by then where to place the transaction in the serial schedule.

Remember: Given a set of transactions there may be several equivalent serial histories. \Rightarrow Scheduler has some freedom of choice. Prefix commit closure guarantees the freedom.

Synchronization strategies

7

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. **during** vs. **at commit** of a TA

Aggressive scheduler: Set equivalence time to time at the beginning \Rightarrow Try immediate execution of an operation \Rightarrow Little freedom for reordering (fewer equivalent serial schedules can be constructed).

Conservative scheduler: As long as equivalence time is open, execution of an operation can be delayed \Rightarrow Some freedom for reordering, but slower progress of some transactions.

Pessimistic strategies
“Deal with conflicts as they arise”

Synchronization strategies

8

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. during vs. at commit of a TA

Optimistic scheduler: That just leaves equivalence time to be set to commit time \Rightarrow No decision until then \Rightarrow Failure possible \Rightarrow Limited opportunities for reordering.

Optimistic strategies
“Let’s hope there are no conflicts”

CSR Safety

Definition 6.1 (CSR Safety):

For a scheduler S , **Gen(S)** denotes the set of all schedules that S can generate. A scheduler is called **CSR safe** if $\text{Gen}(S) \subseteq \text{CSR}$.

Basic locking schedulers

Synchronization strategy pursued

11

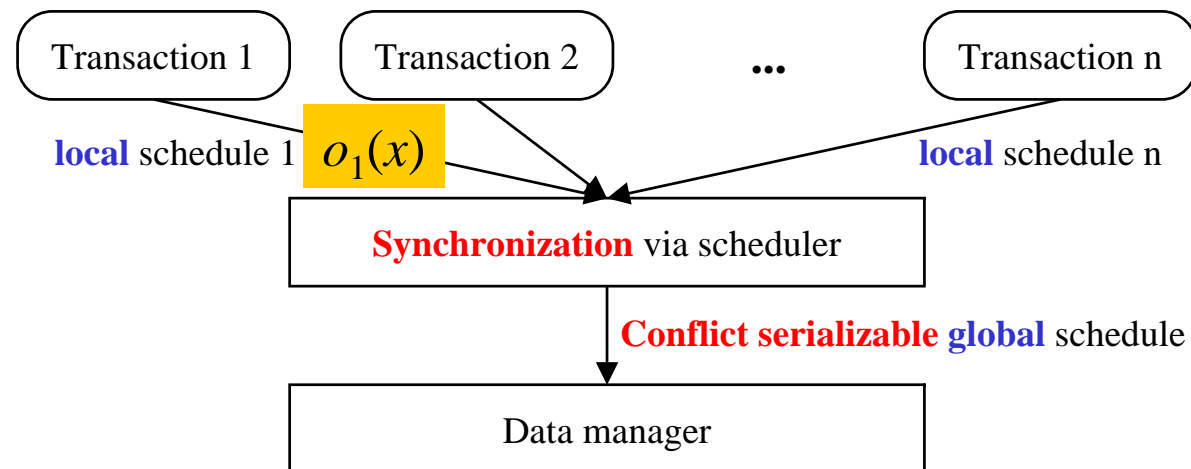
Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. **during** vs. **at commit** of a TA

Conservative scheduler: As long as equivalence time is open, execution of an operation can be delayed \Rightarrow Some freedom for reordering, but slower progress of some transactions.

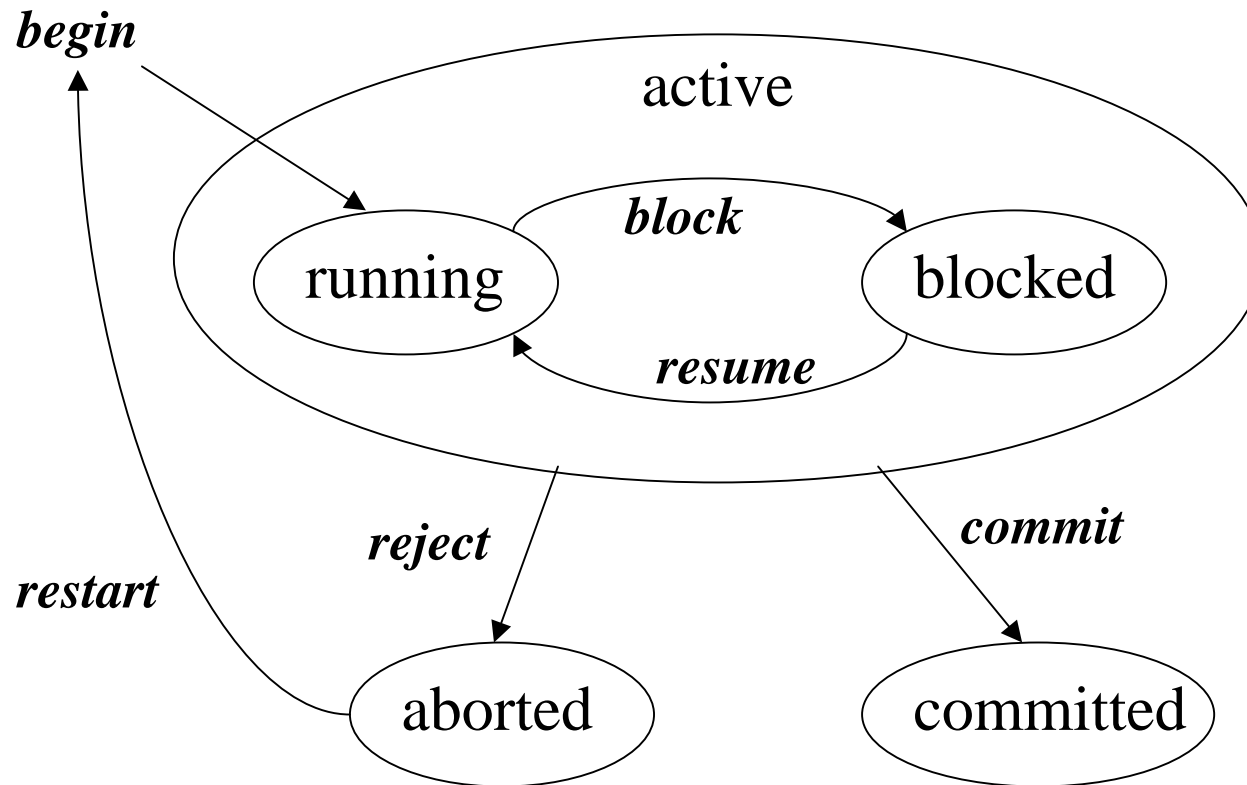
Scheduler options

12



- A scheduler has for each $o_1(x)$ reaching it three options:
 - no conflict** ◆ immediate execution (passing it on to the data manager),
 - conflict** ◆ delay (insertion into a queue),
 - ◆ reject (abort the transaction).

Scheduler Actions and Transaction States



Locks and lock operations

Recognize conflicts by locking:

- Transactions wishing to access a data element solicit a lock for the element.

Operations:

- ◆ $rl_i(x)$: read lock, *more precisely: lock set by a reader* **share lock**
 - ◆ $wl_i(x)$: write lock, *more precisely: lock set by a writer* **exclusive lock**
 - ◆ If p is either read or write we write $pl_i(x)$.
 - ◆ Further: $pl_i(x)$ denotes both, the lock operation and the resulting lock.
- Once the transaction no longer needs the element it unlocks the element: $pu_i(x)$.

Lock compatibility

- While an element remains locked, other transactions *may* not be able to access it and *must* then wait until the element becomes unlocked.

Definition 6.2

Two locks $pl_i(x)$ and $ql_j(y)$ **are in conflict** ($pl_i(x) \nVdash ql_j(y)$)
 $\Leftrightarrow p \nVdash q$ and $i \neq j$ and $x = y$.

Resulting **compatibility matrix**:

		lock holder		
		–	$rl_i(x)$	$wl_i(x)$
lock requestor	$rl_j(x)$	✓	✓	–
	$wl_j(x)$	✓	–	–

Minimal lock rules

To ensure that conflicting operations are ordered:

- **LR1:** Each data operation $p_i(x)$ must be preceded by $pl_i(x)$ and followed by $pu_i(x)$.

Each data element must be locked sometime prior to first access, and unlocked sometime after last access.

- **LR2:** For each x and t_i there is at most one $rl_i(x)$ and at most one $wl_i(x)$.

A transaction can lock the same data element at most once in each mode.

- **LR3:** No $ru_i(x)$ or $wu_i(x)$ is redundant.

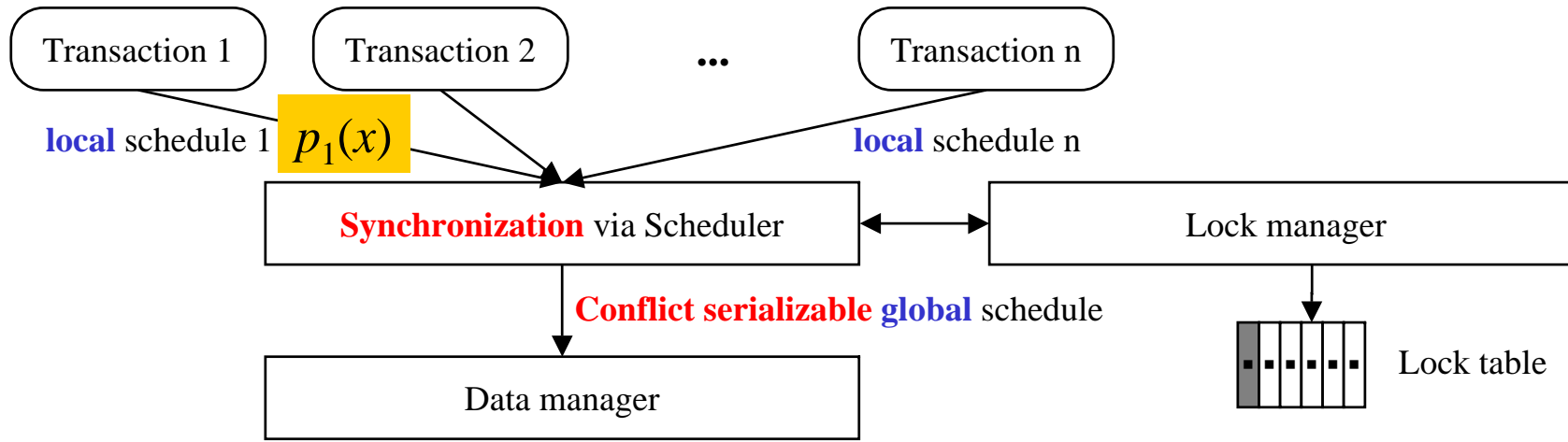
No unnecessary unlocks.

- **LR4:** If x is locked by both t_i and t_j , then these locks are compatible.

No data element is locked in incompatible modes.

Lock processing

17



Lock table holds active locks for each data element.

1. Arrival of $p_i(x)$. Check lock table.
2. Check whether there is $q_l_j(x)$.
 1. no: Set $pl_i(x)$.
 2. yes: Conflict?
 1. no: Set $pl_i(x)$.
 2. yes: transaction i must wait.

Extended Schedules (1)

Example 6.3:

$$t_1 : r_1(x) w_1(y) c_1$$

$$t_2 : w_2(y) w_2(x) c_2$$

Take schedule:

$$r_1(x) w_2(y) w_2(x) c_2 w_1(y) c_1$$

Extended schedule satisfying rules LR1 – LR4 :

$$s_1 = rl_1(x) r_1(x) ru_1(x) wl_2(y) w_2(y) wl_2(x) w_2(x) wu_2(x) wu_2(y) \\ c_2 wl_1(y) w_1(y) wu_1(y) c_1$$

Extended Schedules (2)

Definition 6.4:

Let s be an extended schedule. Then $DT(s)$ refers to the projection of s that excludes the lock operations.

Example 6.5:

$$DT(s_1) = r_1(x) w_2(y) w_2(x) c_2 w_1(y) c_1$$

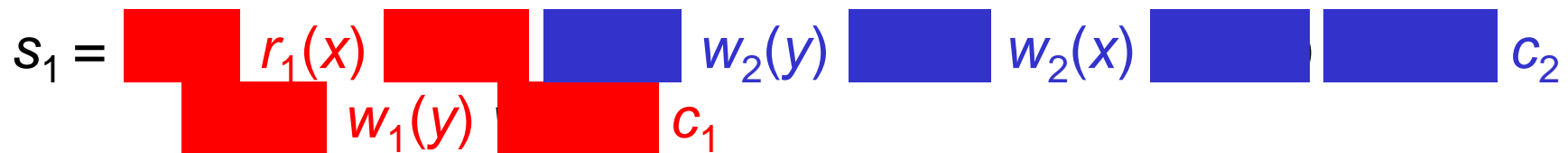
Remark:

If it is clear from the context s will refer to both, s and $DT(s)$.

Two-Phase Locking (2PL) (1)

- Rules LR1-LR4 do not suffice to guarantee conflict serializability!

Example 6.6:



$$DT(s_1) = r_1(x) w_2(y) w_2(x) c_2 w_1(y) c_1$$

Analysis for s_1 :

- s_1 satisfies rules LR1 – LR4.
- $r_1(x) <_{s_1} w_2(x)$ and $w_2(y) <_{s_1} w_1(y) \succ G(CP(DT(s_1)))$ is cyclic
 $\succ s_1$ is not serializable.

Two-Phase Locking (2PL) (2)

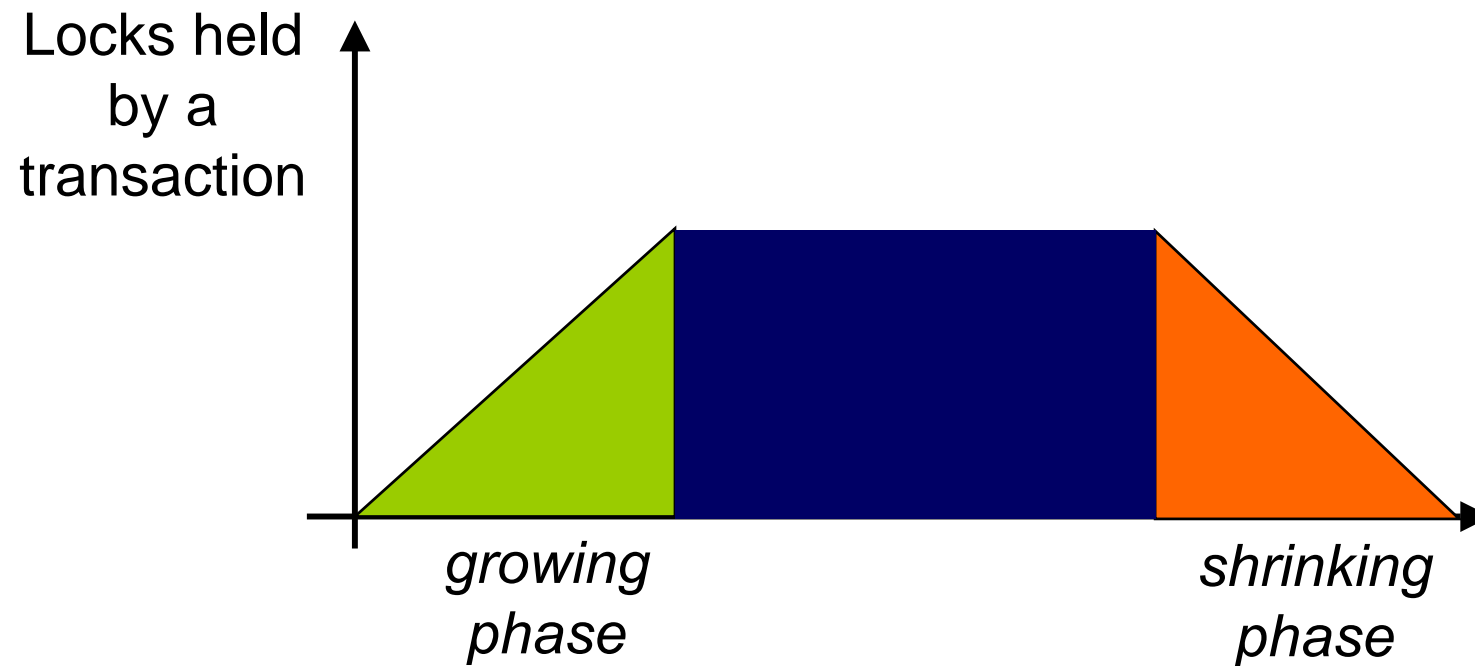
- We add:

Definition 6.6 (2PL):

A locking protocol is **two-phase (2PL)** if for every schedule s and every transaction $t_i \in \text{trans}(s)$ no q_l step follows the first pu_i step ($p, q \in \{r, w\}$).

A transaction quits setting new locks as soon as it issued the first unlock operation.

Two-Phase Locking (2PL) (3)



Two-Phase Locking (2PL) (4)

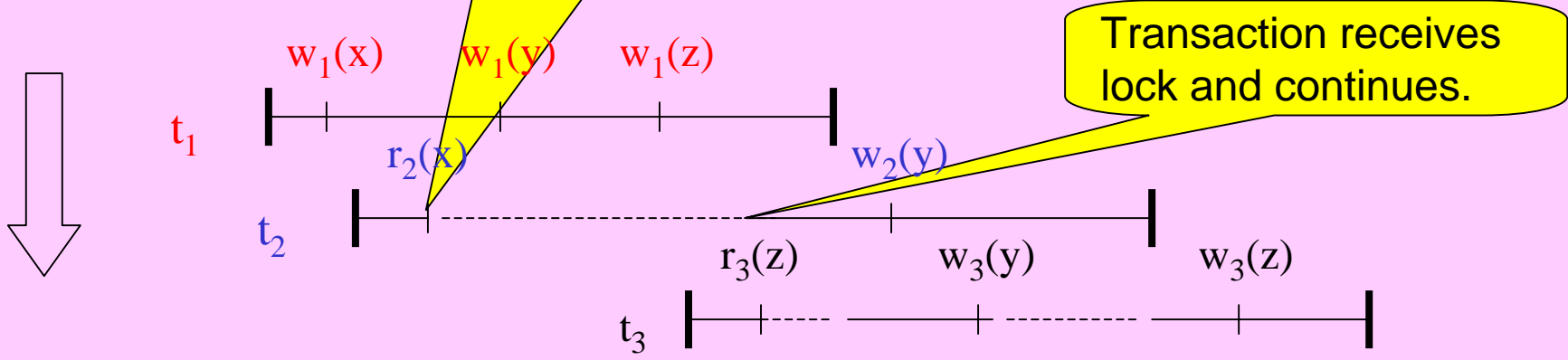
Definition 6.7 (2PL):

A locking protocol is **Two-Phase Locking (2PL)** if for every schedule s and every transaction $t_i \in \text{trans}(s)$, step $(p, q \in \{r, w\})$.

Transaction wishes to execute operation. It requests the lock but is blocked until element becomes accessible.

Example 6.8:

arrival order = $w_1(x) r_2(x) w_1(y) w_1(z) r_3(z) c_1 w_2(y) w_3(y) c_2 w_3(z) c_3$



Transaction receives lock and continues.

$wl_1(x) w_1(x) wl_1(y) w_1(y) wl_1(z) w_1(z) wu_1(x) rl_2(x) r_2(x) wu_1(y) wu_1(z) c_1$
 $rl_3(z) r_3(z) wl_2(y) w_2(y) wu_2(y) ru_2(x) c_2$
 $wl_3(y) w_3(y) wl_3(z) w_3(z) wu_3(z) wu_3(y) c_3$

Correctness proof: Principle

Goal: Prove $Gen(2PL) \subseteq CSR$.

Proof in two steps:

1. Formalize the properties of 2PL-histories.
2. Show that these properties imply conflict serializability.

Correctness proof: Step 1

Lemma 6.9 (Properties of 2PL schedules):

Let s be generated by a 2PL scheduler. Then the following holds for each transaction:

$t_i \in DT(s)$:

- LR1 1. $p_i(x) \in CP(s) \succ pl_i(x), pu_i(x) \in CP(s) \wedge pl_i(x) <_s p_i(x) <_s pu_i(x)$
- LR4 2. $p_i(x), q_j(x) \in CP(s), i \neq j: p_i(x) \not\parallel q_j(x) \succ pu_i(x) <_s ql_j(x) \vee qu_j(x) <_s pl_i(x)$
- 2PL 3. $p_i(x), q_i(y) \in CP(s) \succ pl_i(x) <_s qu_i(y)$
- 4. LR2, LR3 hold.

LR1: Each data operation $p_i(x)$ must be preceded by $pl_i(x)$ and followed by $pu_i(x)$.

LR2: For each x and t_i there is at most one $rl_i(x)$ and at most one $wl_i(x)$.

LR3: No $ru_i(x)$ or $wu_i(x)$ is redundant.

LR4: If x is locked by both t_i and t_j , then these locks are compatible.

Correctness proof: Step 2 (1)

Lemma 6.10 (conflict graph of 2PL schedules):

Let s be generated by a 2PL scheduler and $G := G(CP(DT(s)))$ be the conflict graph of $CP(DT(s))$. Then:

1. $(t_i, t_j) \in E(G) \succ \exists x, p_i(x), q_j(x): pu_i(x) <_s ql_j(x)$
2. (t_1, t_2, \dots, t_n) is path in $G \succ pu_1(x) <_s ql_n(y)$
3. G is acyclic

Correctness proof: Step 2 (2)

1. $(t_i, t_j) \in E(G) \succ \exists x, p_i(x), q_j(x): pu_i(x) <_s ql_j(x)$
2. (t_1, t_2, \dots, t_n) is path in $G \succ pu_1(x) <_s ql_n(y)$
3. G is acyclic

Proofs generally are by giving an interpretation to the edges of the relevant graph. (1) does it here:

Assume $(t_i, t_j) \in E(G) \succ \exists x, p_i(x), q_j(x): p_i(x) \not\parallel q_j(x) \wedge p_i(x) <_s q_j(x)$ (Def. G)

Now, by Lemma 6.9, (1): $pl_i(x), pu_i(x) \in CP(s) \succ pl_i(x) <_s p_i(x) <_s pu_i(x)$

By Lemma 6.9, (2): $i \neq j: p_i(x) \not\parallel q_j(x) \succ pu_i(x) <_s ql_j(x) \vee qu_j(x) <_s pl_i(x)$

case $qu_j(x) <_s pl_i(x) \succ q_j(x) <_s p_i(x)$ (by 6.9(1), contradiction)

leaves case $pu_i(x) <_s ql_j(x) \succ p_i(x) <_s q_j(x)$ (by 6.9(1))

1. $p_i(x) \in CP(s) \succ pl_i(x), pu_i(x) \in CP(s) \wedge pl_i(x) <_s p_i(x) <_s pu_i(x)$.
2. $p_i(x), q_j(x) \in CP(s), i \neq j: p_i(x) \not\parallel q_j(x) \succ pu_i(x) <_s ql_j(x) \vee qu_j(x) <_s pl_i(x)$

Correctness proof: Step 2 (2)

1. $(t_i, t_j) \in E(G) \succ \exists x, p_i(x), q_j(x): pu_i(x) <_s ql_j(x)$
2. (t_1, t_2, \dots, t_n) is path in $G \succ pu_1(x) <_s ql_n(y)$
3. G is acyclic

By induction on n :

$n = 2$: from Lemma 6.10 (1)

$n > 2$: Let Lemma 6.10 (2) be true for $k = n-1$:

1. $\exists p_1(x), o_k(z) \in CP(DT(s)) \quad pu_1(x) <_{CP(s)} ol_k(z)$
With $t_k \rightarrow t_n$ and Lemma 6.10 (1):
2. $\exists o'_k(y), q_n(y) \in CP(DT(s)) \quad o'_k(y) <_{CP(s)} ql_n(y)$
3. $ol_k(z) <_{CP(s)} o'_k(y)$ (Lemma 6.9, (3))
 $\succ pu_1(x) <_{CP(s)} ql_n(y)$ (1, 3 and 2 and transitivity of „ $<_{CP(s)}$ “)

3. $p_i(x), q_i(y) \in CP(s) \succ pl_i(x) <_s qu_i(y)$.

Correctness proof: Step 2 (2)

1. $(t_i, t_j) \in E(G) \succ \exists x, p_i(x), q_j(x): pu_i(x) <_s ql_j(x)$
2. (t_1, t_2, \dots, t_n) is path in $G \succ pu_1(x) <_s ql_n(y)$
3. G is acyclic

Proof by contradiction:

Suppose $G(CP(DT(s)))$ contains cycle $t_1 \rightarrow \dots \rightarrow t_n \rightarrow t_1$ where $n > 1$.

$\exists p_1(x), q_1(y) \in CP(DT(s)) \quad pu_1(x) <_{CP(s)} ql_1(y)$ (with Lemma 6.10 (2))

Contradicts Lemma 6.9 (3).

2PL rule is essential!

3. $p_i(x), q_i(y) \in CP(s) \succ pl_i(x) <_s qu_i(y)$.

Correctness proof: Step 2 (3)

30

Theorem 6.11 (safe)

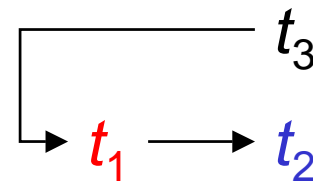
$$\text{Gen}(2PL) \subset \text{CSR}$$

Proof:

\subseteq follows directly from Lemma 6.10

Proper subset:

$$s = w_1(x) r_2(x) c_2 r_3(y) c_3 w_1(y) c_1$$



Two-Phase Locking (2PL) (5)

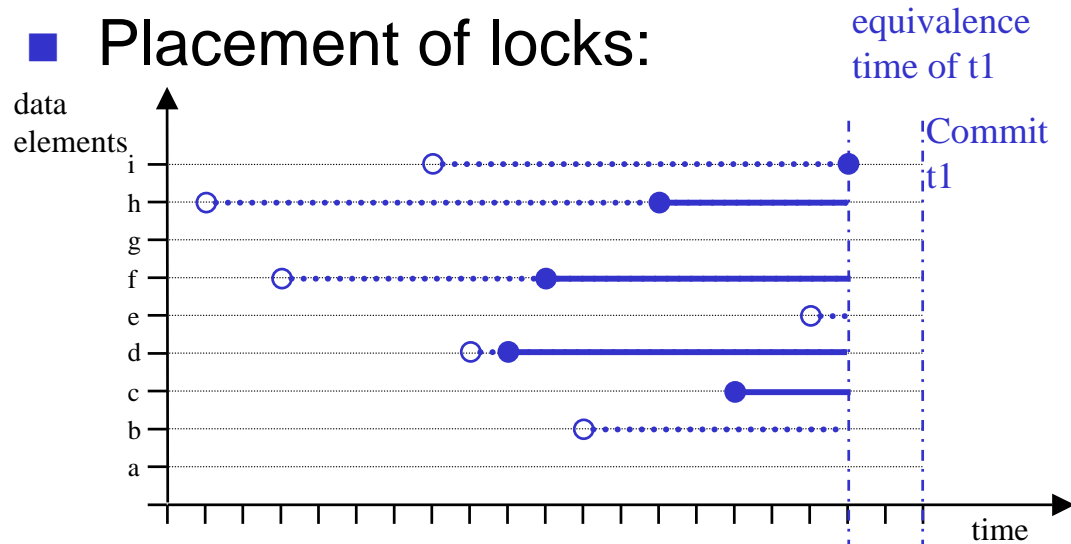
31

- **Remark:**

LR2-LR4 permit that a transaction may issue $w_i(x)$ after $r_i(x)$ and, hence, **escalate** a $r/$ lock to a $w/$ lock. The correctness proof for 2PL is not affected!

2PL: Graphical illustration

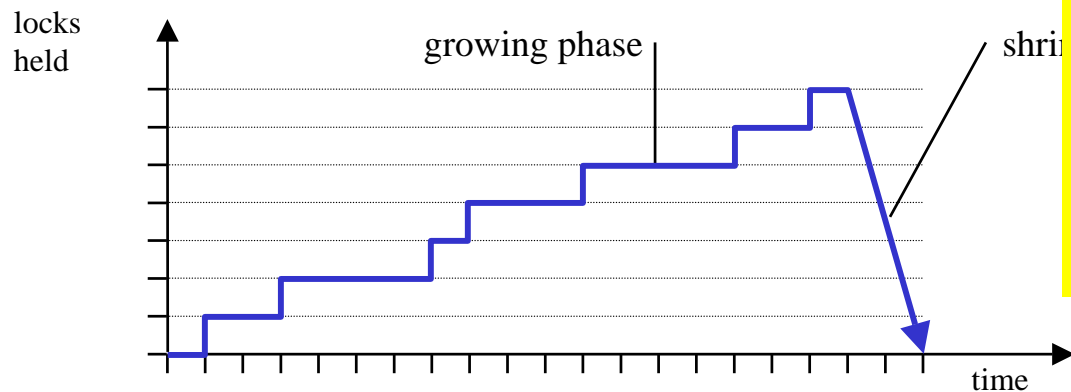
■ Placement of locks:



Legend:

- Read by t1
- Write by t1
- ⋯ Read lock by t1
- Write lock by t1

■ Number of locks held:



Interpretation:

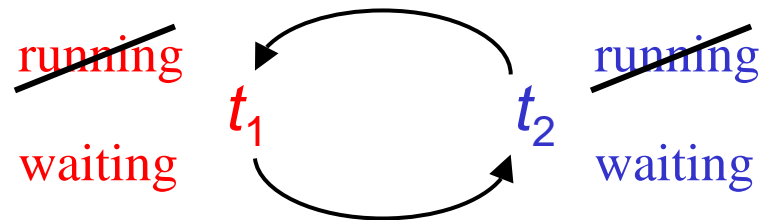
2PL schedule is conflict equivalent to a serial schedule where all transactions are ordered wrt their final time of *maximum lock ownership*.

Deadlocks (1)

What happens if a cycle threatens to develop?

Example 6.12

$S_1 =$
 $w_1(y) \text{ } ru_1(x) \text{ } wu_1(y) \text{ } c_1$
 $w_2(x) \text{ } w_2(x) \text{ } wu_2(x) \text{ } wu_2(y) \text{ } c_2$



Schedule cannot be continued: Both t_1 and t_2 must wait.

Analysis: Attempt to rearrange serial order of transactions.

Fortunately, deadlocks not on every rearrangement!

Deadlocks (2)

34

2PL does not avoid deadlocks!

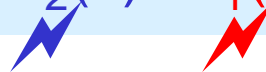
Frequent reason for deadlocks: **lock escalation** (also: **lock conversion**) *rl* to *wl*.

Example 6.13

t_1 : *rl*₁(*x*) *r*₁(*x*) *wl*₁(*x*) *w*₁(*x*) *wu*₁(*x*) *c*₁

t_2 : *rl*₂(*x*) *r*₂(*x*) *wl*₂(*x*) *w*₂(*x*) *wu*₂(*x*) *c*₂

s_1 : *rl*₁(*x*) *r*₁(*x*) *rl*₂(*x*) *r*₂(*x*) *wl*₂(*x*) *wl*₁(*x*) **X**



Deadlock detection

- Via **timeout**:
 - ◆ Each lock request is assigned a maximum wait time.
 - ◆ After expiration a deadlock is assumed.
 - ◆ Simple implementation, but if wait time is badly chosen erroneous abort or belated detection.
- Via **waits-for graph** $WFG = (N, E)$ where
 - $N = active(s)$
 - $E = \{(t_i, t_j) \mid t_i \text{ waits for unlock by } t_j\}$.
 - ◆ Deadlock if WFG has a cycle.
 - ◆ More costly management than for timeouts, but more precise outcome.

Deadlock Resolution

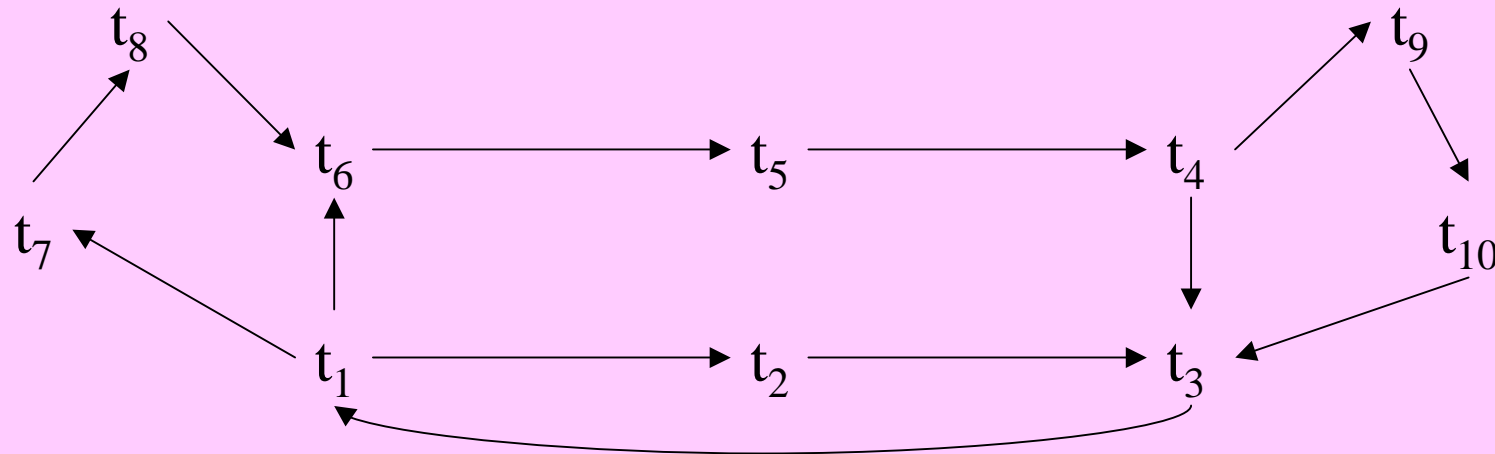
Choose a transaction on a WFG cycle as a **deadlock victim** and abort this transaction, and repeat until no more cycles.

Possible victim selection strategies:

1. Last blocked
2. Random
3. Youngest
4. Minimum locks
5. Minimum work
6. Most cycles
7. Most edges

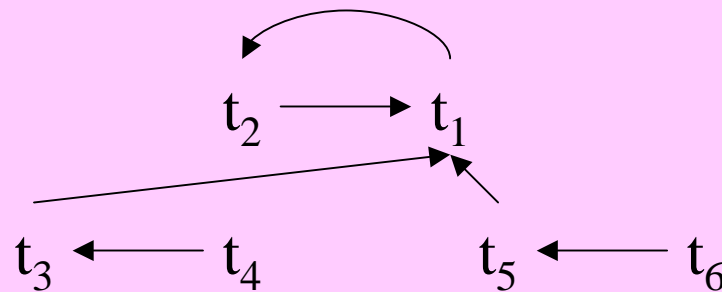
Illustration of Victim Selection Strategies

Example 6.14:



Most-cycles strategy would select t_1 (or t_3) to break all 5 cycles.

Example 6.15:



Most-edges strategy would select t_1 to remove 4 edges.

Deadlock Resolution: Problem

- All aforementioned resolution strategies can result in **livelock** (or **starvation**):

The same transaction is re-selected for abort and restarted (thus again ending up in a cycle and being victimized), etc.

Deadlock Prevention

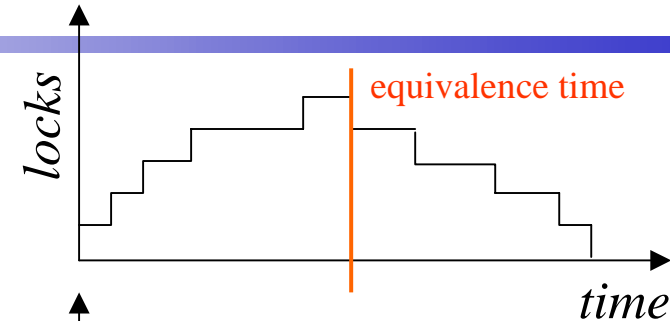
How about avoiding cycles altogether? \Rightarrow **Restrict lock waits** to ensure **acyclic WFG** at all times.

Reasonable deadlock prevention strategies:

- 1. Wait-die:**
upon t_i blocked by t_j :
if t_i started before t_j then wait else abort t_i
Transactions can only be blocked by later ones.
- 2. Wound-wait:**
upon t_i blocked by t_j :
if t_i started before t_j then abort t_j else wait
Transactions can only be blocked by earlier ones.
- 3. Immediate restart:**
upon t_i blocked by t_j : abort t_i
No blocking whatsoever.
- 4. Running priority:**
upon t_i blocked by t_j :
if t_j is itself blocked then abort t_j else wait
Blocked transactions must not impede active ones.

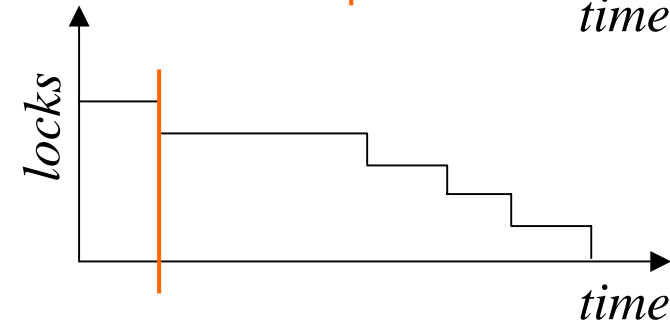
Variants of 2PL

general 2PL



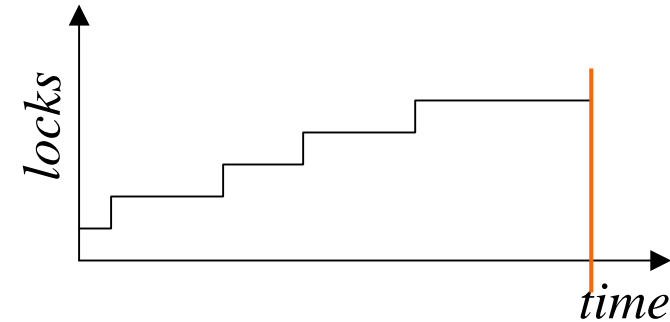
Definition 6.16 (Conservative 2PL):

Under **static or conservative 2PL (C2PL)** each transaction acquires all its locks before the first data operation (**preclaiming**).



Definition 6.17 (Strict 2PL):

Under **strict 2PL (S2PL)** each transaction holds all its write locks until the transaction terminates.

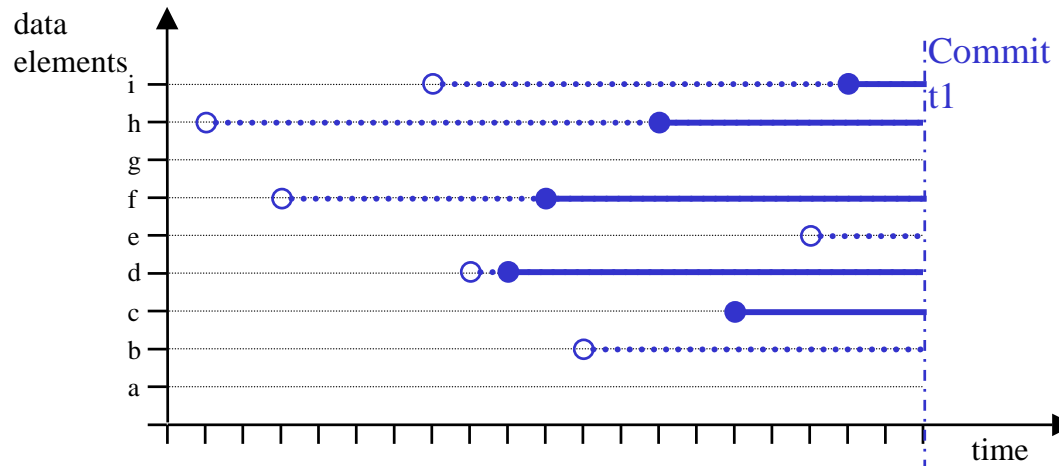


Definition 6.18 (Strong 2PL):

Under **strong 2PL (SS2PL)** each transaction holds all its locks (i.e., both r and w) until the transaction terminates.

SS2PL: Graphical illustration

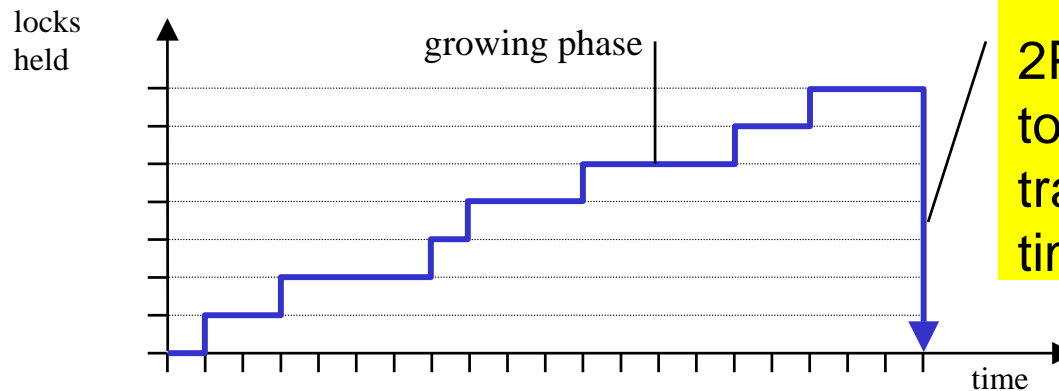
■ Placement of locks:



Legend:

- Read by t1
- Write by t1
- ⋯ Read lock by t1
- Write lock by t1

■ Number of locks held:



Interpretation:

2PL schedule is conflict equivalent to a serial schedule where all transactions are ordered wrt their time of *commit*.

Rigorous Schedules (1)

- SS2PL generates rigorous schedules.
 - ◆ s is **rigorous** if for all transactions t_i, t_j in $CP(s)$ where $i \neq j$: if $o_i(x)$ before $o_j(x)$ in s and $o_i(x)$ in conflict with $o_j(x)$ then c_i before $o_j(x)$.

History h is **commit-ordered** if:

$\forall t_i, t_j \in h, i \neq j, p \in op_i, q \in op_j: (p, q) \in \text{conf}(h) \succ c_i <_h c_j$

Theorem 6.19:

$Gen(SS2PL) \subset COCSR$

Theorem 6.20:

$Gen(SS2PL) \subset Gen(S2PL) \subset Gen(2PL)$

Rigorous Schedules (2)

Remark 6.21:

- SS2PL is the protocol offered by most commercial database systems.
 - ◆ COCSR plays a central role wrt recovery.
 - ◆ Transactions can leave locking/unlocking entirely to the scheduler.

Example

Example 6.22

arrival order = $w_1(x) r_2(x) w_1(y) w_1(z) r_3(z) c_1 w_2(y) w_3(x) c_2 w_3(z) c_3$

2PL $w_1(x) w_1(y) w_1(z) r_2(x) r_3(z) c_1 w_2(y) w_3(x) c_2 w_3(z) c_3$

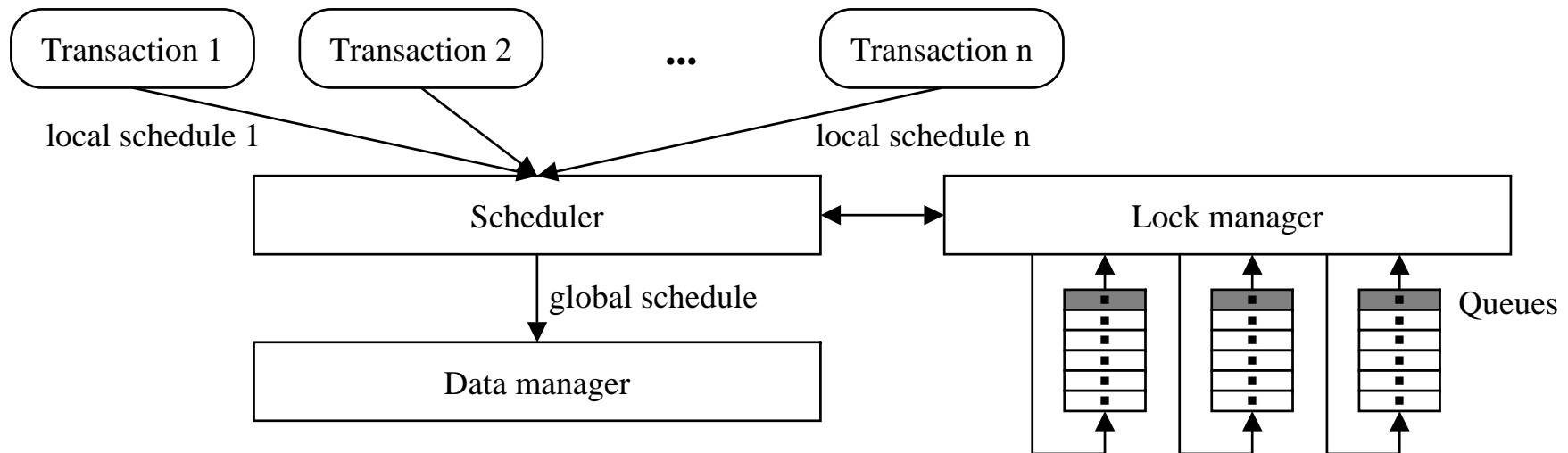
C2PL $w_1(x) w_1(y) r_2(x) w_1(z) c_1 w_2(y) r_3(z) w_3(x) c_2 w_3(z) c_3$

S2PL $w_1(x) w_1(y) w_1(z) c_1 r_2(x) r_3(z) w_2(y) w_3(x) c_2 w_3(z) c_3$

SS2PL $w_1(x) w_1(y) w_1(z) c_1 r_2(x) r_3(z) w_2(y) c_2 w_3(x) w_3(z) c_3$

Implementation of SS2PL (1)

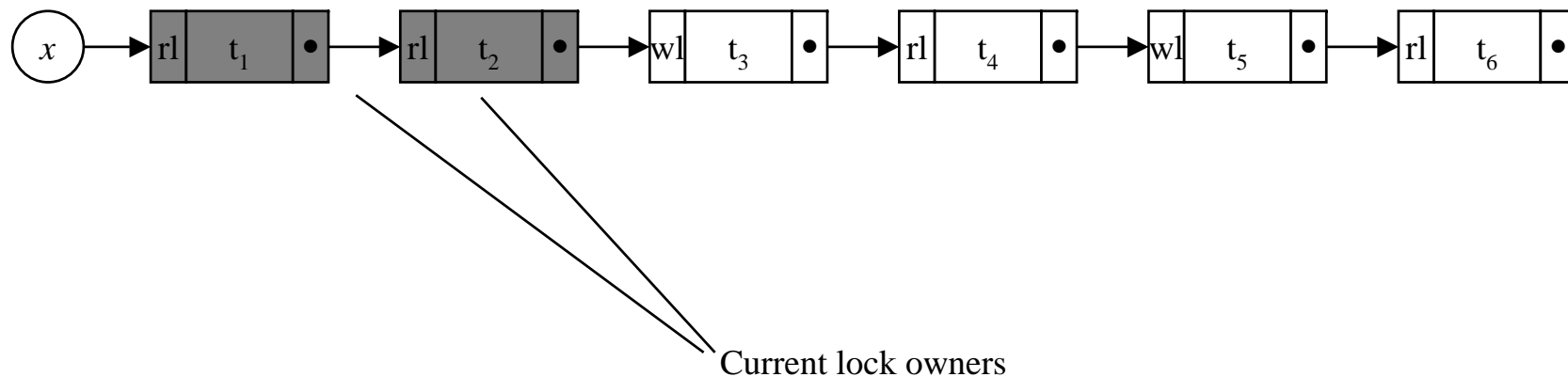
- Refinement of the earlier architecture:



Implementation of SS2PL (2)

Assignment of locks by the lock manager:

- Administer (at least conceptually) a queue for each data element x , where the first queue elements refer to the active owners and the remaining elements to the waiting transactions. If there are no locks on x the queue is empty.



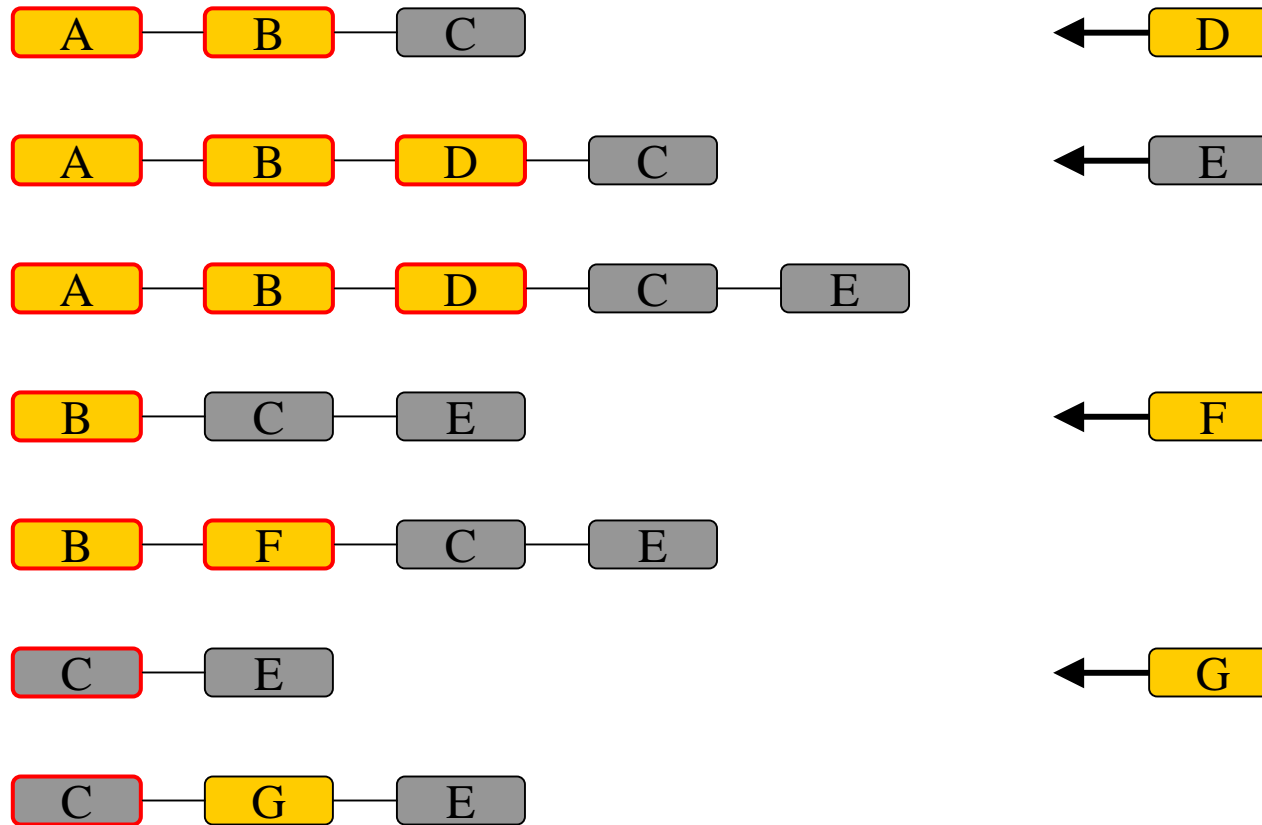
Implementation of SS2PL (3)




- (Optimizing) strategies for satisfying a lock request on x:
 - ◆ **Fair:** Always insert new element at the end of the queue.
 - ◆ **Favoring readers:**
 - A new share lock is inserted in front of all waiting writers. If there is no active writer, no further wait.
 - A new exclusive lock is inserted at the end of the queue.
 - ◆ **Favoring writers:**
 - A new exclusive lock is inserted right behind the last writer or if there is none, right behind the last active reader.
 - A new share lock is inserted at the end of the queue.

Implementation of SS2PL (4)

- **Synchronization algorithm favoring readers:**
 - ◆ Wait for next arriving operation op .
 - ◆ If $op = r_i(x)$ request read (share) lock. Check with lock manager whether another transaction already owns write (exclusive) lock on x . If not grant lock, else wait. Once lock is available execute op .
 - ◆ If $op = w_i(x)$ request write lock. Check with lock manager whether no transaction already owns a lock on x or transaction i already owns share lock on x . If so grant lock, else wait. Once lock is available execute op .
 - ◆ If $op = c_i$ call on the database manager to make persistent all database changes by transaction i , then remove all locks held by transaction i .
 - ◆ If $op = a_i$ call on the database manager to roll back all database changes by transaction i , then remove all locks held by transaction i .

Implementation of SS2PL (5)



 reader  writer  active transaction

Multiple Granularity Locking Schedulers

Phantome (1)

Hr. Müller fragt im Rahmen einer Inventur die Bestände der einzelnen Sorten bei der Relation Weißweine ab und errechnet den Gesamtbestand. Hr. Schmidt erweitert unterdessen das Sortiment um 10 Kisten Grauburgunder.

Weißweine	
Artikel	Bestand
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller-Thurgau	100

Bestände	
Sorte	Bestand
Weißweine	159
Rotweine	200

Phantome (2)

Weißweine	
Artikel	Bestand
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller-Thurgau	100

t_M

zur Zeit 10: t_S fügt ein:
(Grauburgunder, 10)

zur Zeit 9: Position von t_M

zur Zeit 11: t_S ändert:
(Weißweine, 169)

zur Zeit 12: Inkonsistente
Sicht von t_M : $159 \neq 169$

Bestände	
Sorte	Bestand
Weißweine	159
Rotweine	200

s: $rl_M(Gu) r_M(Gu) rl_M(Ri) r_M(Ri) rl_M(Si) r_M(Si) rl_M(Wb) r_M(Wb) wl_S(Gb) w_S(Gb)$
 $wl_S(Ww) w_S(Ww) wu_S(Gb) wu_S(Ww) c_S rl_M(Ww) r_M(Ww) \dots$

Phantome (3)

s erfüllt 2PL!

s: $rl_M(Gu) r_M(Gu) rl_M(Ri) r_M(Ri) rl_M(Si) r_M(Si) rl_M(Wb) r_M(Wb) wl_S(Gb) w_S(Gb)$
 $wl_S(Ww) w_S(Ww) wu_S(Gb) wu_S(Ww) c_S rl_M(Ww) r_M(Ww) \dots$

DT(s): $r_M(Gu) r_M(Ri) r_M(Si) r_M(Wb) w_S(Gb) w_S(Ww) c_S r_M(Ww) \dots$
 $t_S \rightarrow t_M$

$DT(s) \in CSR$: Dabei scheint non-repeatable read vorzuliegen! Aber: Alle gelesenen Zustände sind zum gleichen Zeitpunkt c_S gültig.

Phantome (4)

- Worst-case Annahme „in einer Transaktion beeinflussen alle vorangehenden Lesen ein Schreiben“ reicht nicht aus!
- Hier: Auch zwischen Lesen einer Transaktion kann es Zusammenhänge geben.
 - ◆ Diese sind Teil der Transaktionssemantik (hier das zwischenzeitliche Aufsummieren) und werden durch das Transaktionsmodell nicht erfasst.
 - ◆ Wird zum Problem, wenn der Zusammenhang verschiedene Gültigkeitszeitpunkte überbrückt.
- **Phantom**: „Geisterhaftes“, für eine fremde Transaktion nicht sichtbares Datenelement als Verursacher eines fehlerhaften Zusammenhanges.

Phantome (5)

Analyse aus Sicht 2PL: Ein nicht vorhandenes Element kann nicht mit einer Sperre belegt werden.

Abhilfe: Setzen einer Sperre, die das neue Element automatisch mit abdeckt.

Korngröße der Sperre:

- Im Beispiel ist Korngröße das Tupel. Man wähle als nächstgrößeres Korn die Relation.
- Wir betrachten nachfolgend ein Verfahren, das die Wahl der Körnigkeit systematisiert.

Multi-Korngrößen-Sperren (1)

56

Multiple Granularity Locking (MGL):

Ausnutzen, dass in einer Datenbasis von der Struktur her verschiedene Granulate (Korngrößen) erkennbar sind.

Beispiele:

Physische Ebene

(ganze) Datenbasis

Segment, Area, DB-Space

Datei

Seite

Record

Semantische Ebene

Objektbasis

Schema

(Objekttyp-)Extension, Relation

Objekt, Tupel

Attribut

Multi-Korngrößen-Sperren (2)

Nutzeffekte:

1. Kontrolle der Phantome.
2. Kontrolle des Nebenläufigkeitsgrades und des Verwaltungsaufwandes:
 - ◆ Großes Korn \succ wenige Sperren \succ geringe Nebenläufigkeit.
 - ◆ Kleines Korn \succ viele Sperren \succ hohe Nebenläufigkeit.

Multi-Korngrößen-Sperren (3)

- **Beispiele:**
 - ◆ t_1 : Gehaltserhöhung aller Angestellten nach Tarifrunde (Schreiben in alle Angestellten-Objekte)
 - ◆ t_2 : Berechnen der Gesamtpersonalkosten (Lesen aller Angestellten-Objekte)
 - ◆ t_3 : Auswertung eines Wettbewerbs, bei dem die besten 3 Vertreter eine Prämie bekommen (Lesen aller Angestellten-Objekte, Schreiben von 3 Objekten)
- All diese längeren Transaktionen haben eine größere Chance durchzukommen, wenn sie nur eine Sperre auf der Menge aller Angestellten setzen.
- Dann aber: Nur serieller Ablauf möglich.

Absichtssperren (1)

Arten von Sperren:

rl: share lock: Sperrt Korn zum Lesen

wl: exclusive lock: Sperrt Korn zum Schreiben.

irl: intention share lock: Zeigt an, dass (ein) Element(e) innerhalb dieses Korns gelesen werden soll(en). Die *share*-Sperren für die Elemente müssen noch einzeln angefordert werden.

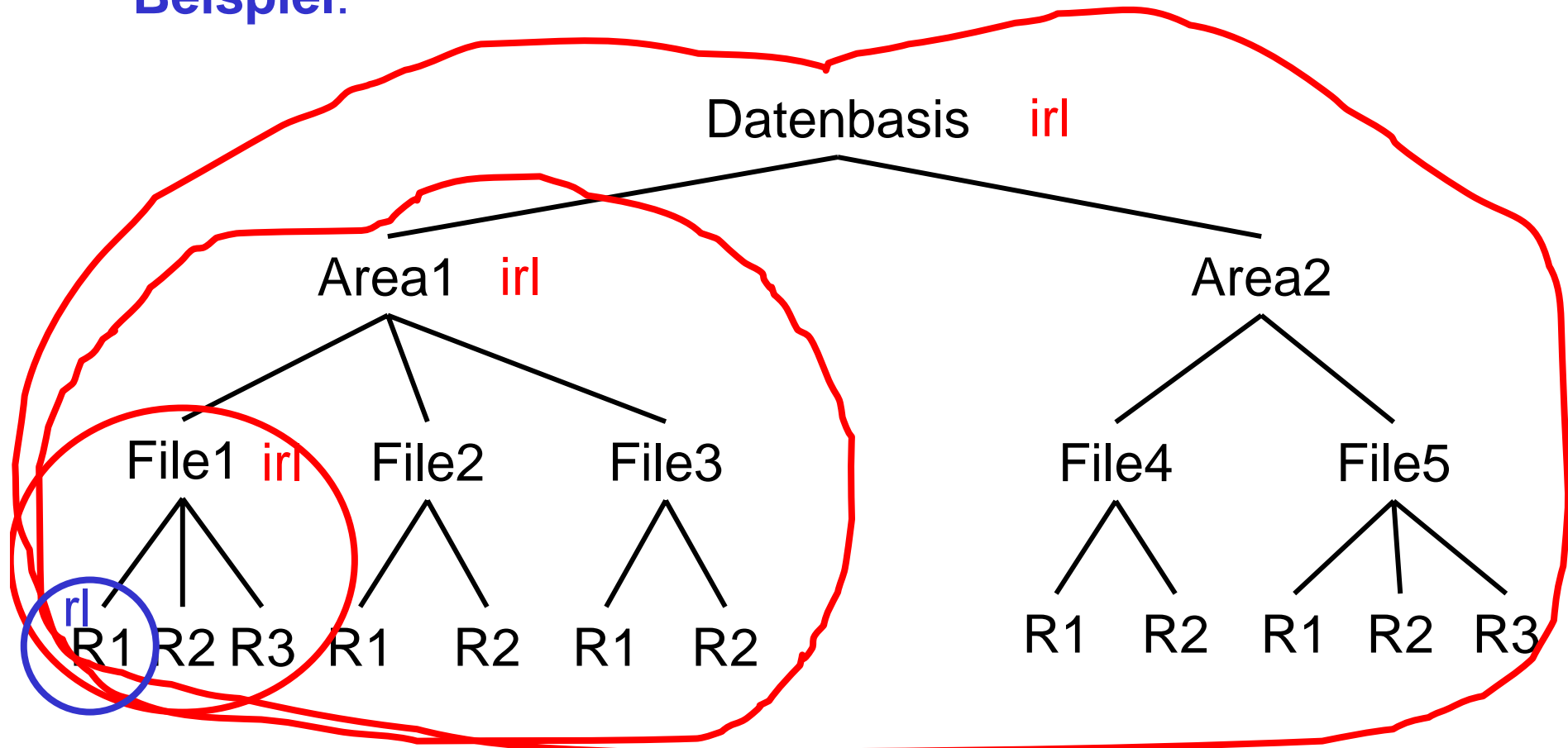
iwl: intention exclusive lock: Analog zum intention share lock.

riwl: share intention exclusive lock: Zeigt an, dass alle Elemente innerhalb dieses Korns gelesen werden und einige geschrieben werden sollen. Die *exclusive*-Sperren zum Schreiben müssen noch explizit angefordert werden.

Absichtssperren (2)

Eine Sperre auf ein Korn X sperrt *implizit* alle unter X befindlichen Körner in gleicher Weise.

Beispiel:



MGL-2-Phasen-Sperrprotokoll (1)

61

Basis-2PL-Protokoll: **Anpassung**

1. Eintreffen von $p_i(x)$: Falls $p_i(x)$ nicht mit einer bereits vergebenen Sperre in Konflikt steht, wird $pl_i(x)$ an t_i vergeben und $p_i(x)$ ausgeführt. Sonst wird $p_i(x)$ verzögert bis die entsprechende Sperre freigegeben ist.
2. Für von t_i bearbeitete x kommt genau ein Schritt $rl_i(x)$ bzw. $wl_i(x)$ derselben Art werden höchstens einmal

- Falls eine r - oder ir -Sperre benötigt wird für ein Korn, müssen alle Vorgänger mit ir oder iw gesperrt sein.
- Falls eine w - oder iw -Sperre benötigt wird für ein Korn, müssen alle Vorgänger mit riw oder iw gesperrt sein.
- Falls t_i ein Datenelement lesen (schreiben) will, so benötigt es eine r - oder riw - (w -) Sperre für das Datenelement oder einen Vorgänger.

MGL-2-Phasen-Sperrprotokoll (2)

Basis-2PL-Protokoll: **Anpassung**

1. Eintreffen von $p_i(x)$: Falls $p_i(x)$ nicht mit einer bereits vergebenen Sperre in Konflikt steht, wird $pl_i(x)$ an t_i vergeben und $p_i(x)$ ausgeführt. Eine Absichts-Sperre kann nur freigegeben werden, falls keine Sperre mehr für Nachfolger gehalten wird.
2. Für jedes von x kommt genau ein Schritt $ri_i(x)$ bzw. $wl_i(x)$ vor. Schritte derselben Art werden höchstens einmal ausgeführt.
3. Eine Sperre $pl_i(x)$ wird nicht eher freigegeben bis nicht wenigstens $p_i(x)$ ausgeführt wurde. Alle Sperren werden bzw. sind zu Transaktionsende freigegeben.
4. Nachdem für eine Transaktion mindestens eine Sperre freigegeben wurde, dürfen keine Sperren mehr an diese Transaktion vergeben werden.

MGL-2-Phasen-Sperrprotokoll (3)

Basis-2PL-Protokoll: **Anpassung**

1. Eintreffen von $p_i(x)$: Falls $p_i(x)$ nicht mit einer bereits vergebenen Sperre in Konflikt steht, wird $pl_i(x)$ an t_i vergeben und $p_i(x)$ ausgeführt. Sonst wird $p_i(x)$ verzögert bis die entsprechende Sperre freigegeben ist.

2. Für jedes von t_i bearbeitete x kommt genau ein Schritt $rl(x)$ bzw. $wl_i(x)$ vor (Sperre gesetzt).

3. Eine Sperre wird freigegeben, wenn alle Transaktionen, die sie halten, abgeschlossen sind zu Transaktion x .

4. Nachdem freigegeben, wird die Sperre für x freigegeben.

gehalten→ angefordert↓	rl	wl	irl	iwl	riwl
rl	yes	no	yes	no	no
wl	no	no	no	no	no
irl	yes	no	yes	yes	yes
iwl	no	no	yes	yes	no
riwl	no	no	yes	no	no

MGL-2-Phasen-Sperrprotokoll (4)

Definition 6.23

MGL-2PL = Basis-2PL + folgende Ergänzungen:

1. Falls eine r - oder ir -Sperrung benötigt wird für ein Korn, müssen alle Vorgänger ir oder iw gesperrt sein.
2. Falls eine w - oder iw -Sperrung benötigt wird für ein Korn, müssen alle Vorgänger mit riw oder iw gesperrt sein.
3. Falls t_i ein Datenelement lesen (schreiben) will, so benötigt es eine r - oder riw - (w -) Sperrung für das Datenelement oder einen Vorgänger.
4. Eine Absichts-Sperrung kann nur freigegeben werden, falls keine Sperrung mehr für Nachfolger gehalten wird.

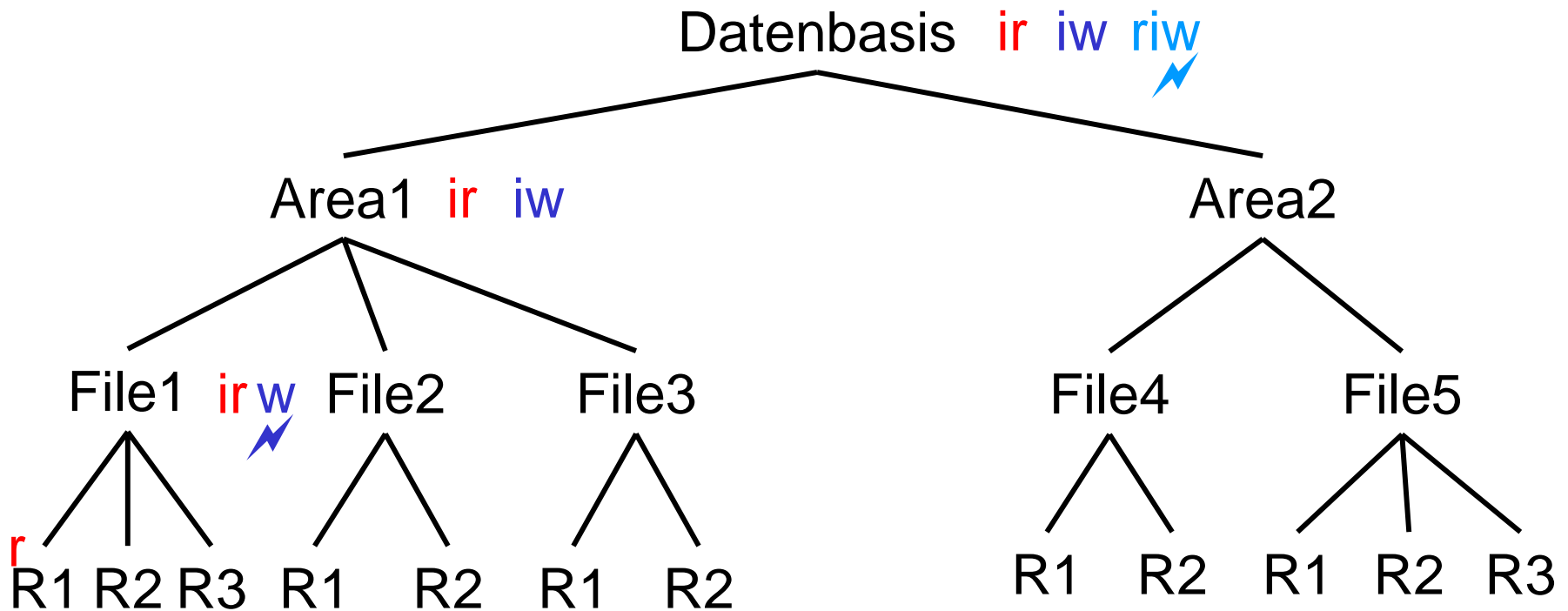
MGL-2-Phasen-Sperrprotokoll (5)

Umsetzung für einzelne Transaktion:

- Eine Transaktion setzt, von der Wurzel her kommend, *ir*-Sperrern bzw. *iw*-Sperrern so lange, bis sie eine *r*- bzw. *w*-Sperrere setzt. Sie setzt *riw*-Sperrern so lange, bis sie eine *w*-Sperrere setzt.
- Eine Transaktion löst ihre Sperrern von den *r*- bzw. *w*-Sperrern her kommend sukzessive bis zur Wurzel.

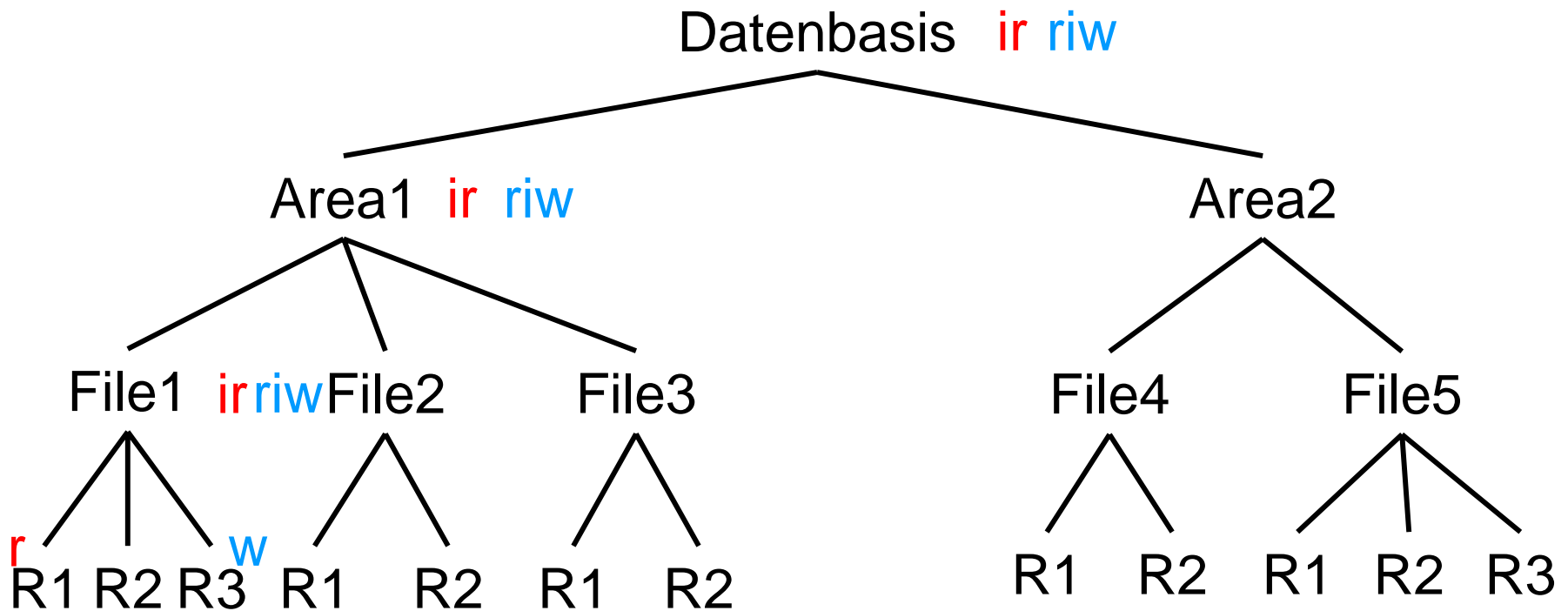
MGL-2-Phasen-Sperrprotokoll (6)

t_1 t_2 t_3



MGL-2-Phasen-Sperrprotokoll (7)

t_1 t_3



Korrektheit (1)

Satz 6.24

Falls alle Transaktionen dem MGL-Protokoll gehorchen, so halten keine zwei Transaktionen zwei in Konflikt stehende Sperren.

Korrektheit (2)

Beweis:

Es genügt, den Satz für Blattknoten zu zeigen, da Beweisführung Vorgänger einbezieht.

Wir nehmen also an, dass t_i und t_j zwei in Konflikt stehende Sperren auf den Blattknoten x halten. Es sind 7 Fälle zu unterscheiden:

	t_1	t_2
1	implizite r Sperre	explizite w Sperre
2	implizite r Sperre	implizite w Sperre
3	explizite r Sperre	explizite w Sperre
4	explizite r Sperre	implizite w Sperre
5	implizite w Sperre	explizite w Sperre
6	implizite w Sperre	implizite w Sperre
7	explizite w Sperre	explizite w Sperre

Korrektheit (3)

Wegen Regel 3 des MGL Protokolls: t_1 hält $rl_1(y)$ für einen Vorgänger y von x .
 Wegen Regel 2 des MGL Protokolls: t_2 hält $iwl_2(z)$ für jeden Vorgänger z von x .
 Insbesondere hält t_2 also $iwl_2(y)$. Widerspruch.

direkter Widerspruch

ähnlich 1

ähnlich 1

direkter Widerspruch

	t_1	t_2
1	implizite r Sperre	explizite w Sperre
2	implizite r Sperre	implizite w Sperre
3	explizite r Sperre	explizite w Sperre
4	explizite r Sperre	implizite w Sperre
5	implizite w Sperre	explizite w Sperre
6	implizite w Sperre	implizite w Sperre
7	explizite w Sperre	explizite w Sperre

Korrektheit (4)

Wg. Regel 3 des MGL Protokolls: t_1 hält $rl_1(y)$ für einen Vorgänger y von x .

Wg. Regel 3 des MGL Protokolls: t_2 hält $wl_2(y')$ für einen Vorgänger y' von x .

Wir unterscheiden drei Unterfälle ('>': Baumordnung):

$y=y'$ Widerspruch.

$y>y'$ Widerspruch, da t_2 $iwl_2(y)$ halten muss. (Konflikt zu $rl_1(y)$)

$y'>y$ Widerspruch, da t_1 $irl_1(y')$ halten muss. (Konflikt zu $wl_2(y')$)

	t_1	t_2
1	implizite r Sperre	explizite w Sperre
2	implizite r Sperre	implizite w Sperre
3	explizite r Sperre	explizite w Sperre
4	explizite r Sperre	implizite w Sperre
5	implizite w Sperre	explizite w Sperre
6	implizite w Sperre	implizite w Sperre
7	explizite w Sperre	explizite w Sperre

ähnlich 2

ung (1)

gehalten→ angefordert↓	rl	wl	irl	iwl	riwl
rl	yes	no	yes	no	no
wl	no	no	no	no	no
irl	yes	no	yes	yes	yes
iwl	no	no	yes	yes	no
riwl	no	no	yes	no	no

... wird für jeden Knoten eine
Eine hinzukommende Sperre
ung, eine wegfallende Sperre
hren (**Sperrkonversion**).

Sperrverschärfung mittels Sperrkonversionstabelle: Falls eine Sperre x gehalten wird und eine Sperre y gewährt wird, ist der Eintrag das Maximum der beiden Sperren:

gehalten→ angefordert↓	r	w	ir	iw	riw
rl	r	w	r	iw	riw
wl	r	w	ir	iw	riw
irl	r	w	ir	iw	riw
iwl	r	w	iw	iw	riw
riwl	r	w	riw	iw	riw

Umsetzung (2)

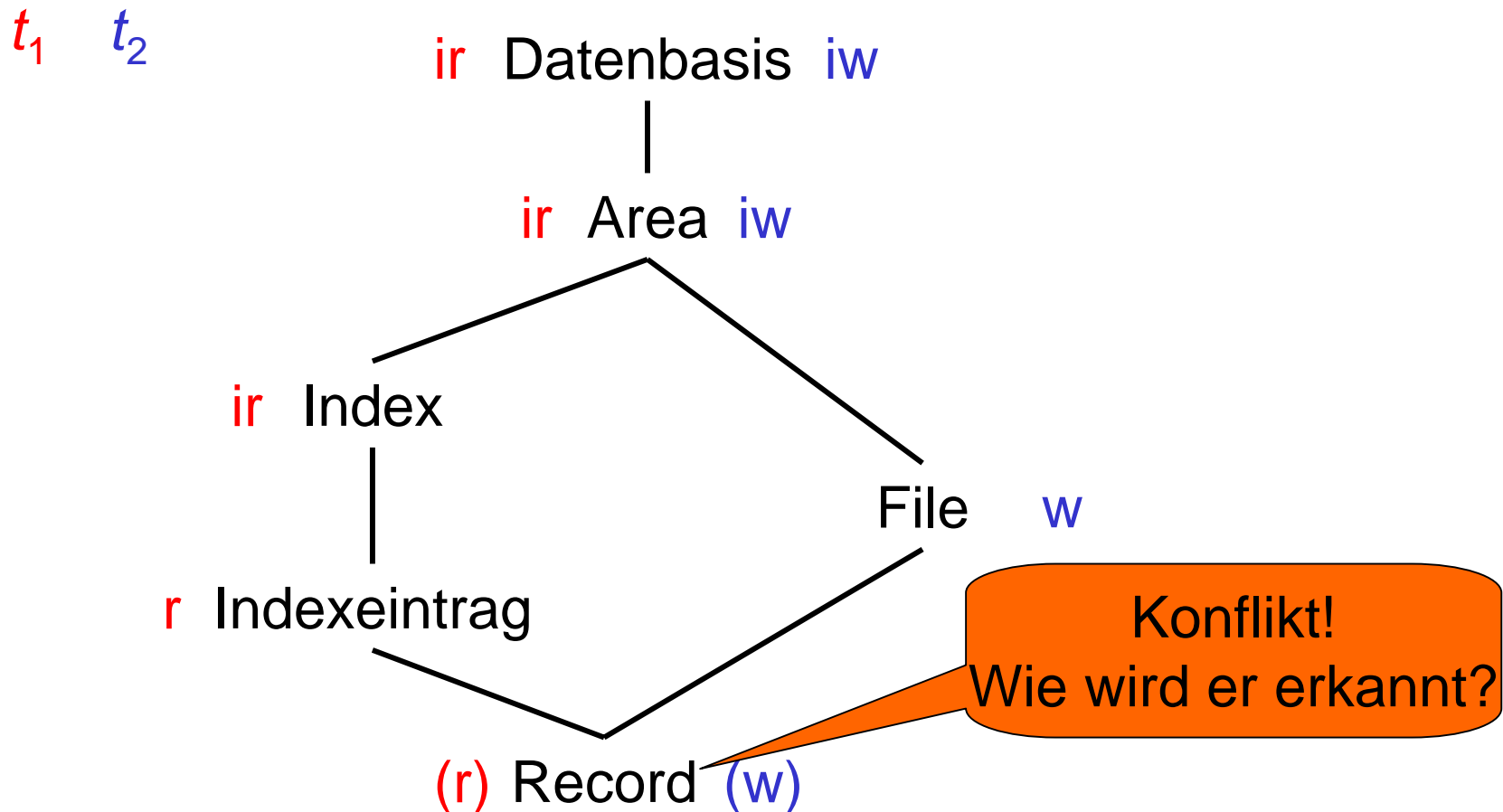
73

Wann wird w - oder r - Sperre auf welchem Korn angefordert?

- Das ist ein Aufwandsproblem. Bei kleiner Korngröße ergeben sich keine Spielräume. Bei großer ist ein gewisses Wissen notwendig darüber, auf wie viele Datenelemente kleinerer Größe zugegriffen wird.
- Dynamische Lösung: Falls eine bestimmte Anzahl von Zugriffen überschritten wird, wird eine entsprechende Sperre für das nächst höhere Korn angefordert.

Körnigkeitsgraphen (1)

Erweiterung von Bäumen auf DAGs.
Beispiel:



Körnigkeitsgraphen (2)

75

1. Falls eine r - oder ir -Sperrung benötigt wird für ein Granulat, müssen alle Vorgänger ir oder iw gesperrt sein.
2. Falls eine w - oder iw -Sperrung benötigt wird für ein Granulat, müssen alle Vorgänger mit riw oder iw gesperrt sein.
- 3a Falls eine Transaktion t_i ein Datenelement x (implizit oder explizit) lesen will, muss sie eine r -, riw - oder w -Sperrung für x oder einen beliebigen Vorgänger von x halten.
- 3b Falls eine Transaktion t_i ein Datenelement x (implizit oder explizit) schreiben will, muss sie auf **jedem** Pfad von x zur Wurzel für einen beliebigen Vorgänger von x eine w -Sperrung halten.
4. Eine ix -Sperrung kann nur freigegeben werden, falls keine Sperrung mehr für Nachfolger gehalten wird.

Tree Locking Schedulers

Baum-Sperr-Protokolle

77

- **Zur Erinnerung:** R/W–Modell basiert allein auf äußerer Beobachtung der Transaktionen.
- **Erwartung:** Zusatzkenntnisse über die Transaktionen lassen sich für höhere Nebenläufigkeit und/oder geringere Berechnungskomplexität nutzen.
- **Beispiel:** Transaktionen mit Durchlauf von Bäumen. Weiß man um
 - ◆ die Struktur des Baumes,
 - ◆ den Einstieg an der Wurzel,kann man Aussagen über die Abfolge von Zugriffen machen.
- **Konsequenz:** Keine 2PL-Forderung notwendig.

Einfaches Baumsperrverfahren (1)

78

- Gegeben Datenbaum mit Wurzel r .
- Es gibt nur eine Zugriffsfunktion $a_i(x)$: Transaktion t_i greift auf Datenelement x , hier ein Knoten eines Baumes, zu.
 - ◆ Daher der Name **Write-only Tree Locking (WTL)**
- Die zugehörige Sperranforderung ist $al_i(x)$. Sperrfreigabe wird durch $au_i(x)$ bezeichnet.
- Falls $i \neq j$, gilt $a_i(x) \not\parallel a_j(x)$ und somit $al_i(x) \not\parallel al_j(x)$.

Einfaches Bausperrverfahren (2)

79

Einfaches Bausperrprotokoll (WTL):

Es müssen die Regeln LR 1 – LR 4 eingehalten werden. Zusätzlich gelten die folgenden Regeln:

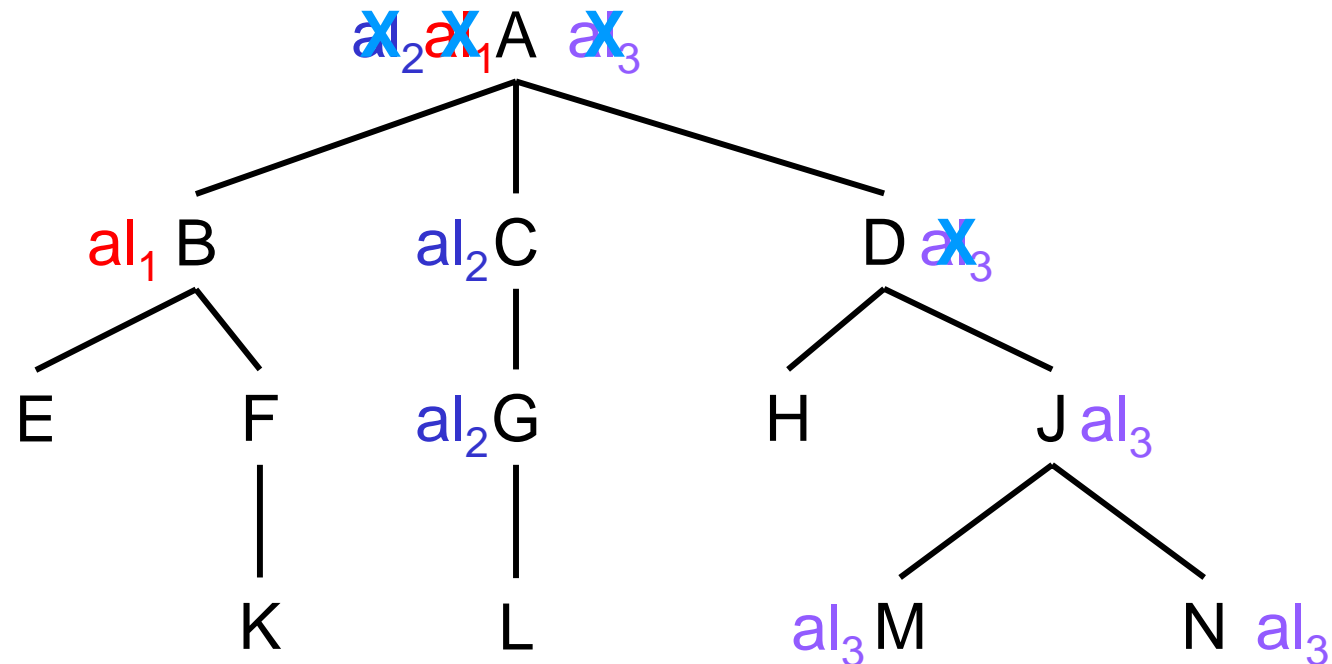
WTL1: Für alle Knoten x außer der Wurzel gilt: $al_i(x)$ kann nur gesetzt werden, wenn $t_i al_i(y)$ hält für y Vaterknoten von x .

WTL2: Nachdem eine Transaktion eine Sperre auf x freigegeben hat, darf sie sie nicht neu setzen.

- **Sperrkopplung (lock coupling):** Der Scheduler kann $al_i(x)$ nur freigeben, falls die Transaktion alle Sperren auf den benötigten Söhnen von x bereits hält.

- Impliziert eine Sperranforderungsrichtung von der Wurzel zu den Blättern.

Einfaches Baumsperrverfahren (3)



Mindestens 1 Sperre pro benutzten Pfad wird garantiert:
Keine Überholung durch andere Transaktion möglich.

Korrektheit

Satz 6.25

$Gen(WTL) \subseteq CSR$

Beweis:

Betrachte $t_i \rightarrow t_j \in G(h)$ für WTL-Historie h .

$\succ \exists a_i(x), a_j(x) \in h \quad a_i(x) \not\parallel a_j(x) \wedge a_i(x) <_h a_j(x)$ (Interpretation Kante!)

$\succ au_i(x) <_h al_j(x)$

$\succ au_i(r) <_h al_j(r)$ (r Wurzel)

WTL1: $al_i(r) <_h al_j(x)$

$\succ au_i(r) <_h al_j(x)$

Annahme: $G(h)$ hat Zyklus der Form

$$t_1 \rightarrow \dots \rightarrow t_n \rightarrow t_1$$

$\succ au_1(r) <_h al_1(r)$ Widerspruch zu WTL2.

Verklemmungsfreiheit

Satz 6.26

Das WTL-Protokoll vermeidet Verklemmungen.

Beweis:

Falls t_i auf die Wurzelsperre wartet, so kann sie nicht in einer Verklemmung involviert sein, da sie keine Sperren besitzt.

Annahme: t_i wartet auf $au_j(x)$, $x \neq r \succ au_j(r) <_h al_i(r)$

Durch Induktion: Falls der Wartegraph einen Zyklus hat, der t_i beinhaltet, so $au_i(r) <_h al_i(r)$. Widerspruch!

Allgemeines Bausperrverfahren (1)

- Differenzierung nach Lesen und Schreiben.
- Falls Transaktionen jeweils nur Lesesperren oder nur Schreibsperrern anfordern, genügen die normalen Konfliktregeln zwischen diesen Sperrern, um Serialisierbarkeit zu gewährleisten.
- Dies ist nicht der Fall, falls eine Transaktion sowohl Lese- als auch Schreibsperrern setzt.
 - **Read-Write Tree Locking (RWTL)**

Allgemeines Baumsperrverfahren (2)

84

Problem: t_i locks root before t_j does,
but t_j passes t_i within a “read zone”

Example

$t_1 = t_2 = w(x) r(y) w(z)$

$h =$ $wl_1(x)$ $w_1(x)$ $rl_1(y)$ $wu_1(x)$ $wl_2(x)$ $w_2(x)$ $rl_2(y)$ $wu_2(x)$ $r_2(y)$ $wl_2(z)$
 $ru_2(y)$ $w_2(z)$ $wu_2(z)$ $r_1(y)$ $wl_1(z)$ $ru_1(y)$ $w_1(z)$ $wu_1(z)$

x
↓
y
↓
z

→ appears to follow WTL rules
but \notin CSR

→ t_1 passes t_2 on element y which both only read

Solution: formalize “read zone”
and enforce two-phase property on “read zones”

Locking Rules of RWTL

For transaction t with read set $RS(t)$ and write set $WS(t)$

let C_1, \dots, C_m be the connected components of $RS(t)$.

A **pitfall** of t is a set of the form

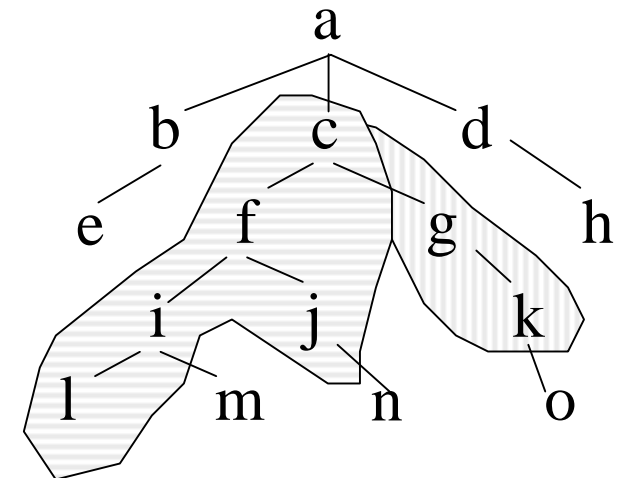
$C_i \cup \{x \in WS(t) \mid x \text{ is a child or parent of some } y \in C_i\}$.

Example:

t with $RS(t)=\{f, i, g\}$ and $WS(t)=\{c, l, j, k, o\}$

has $C_1=\{f, i\}$ and $C_2=\{g\}$

and pitfalls $pf_1=\{c, f, i, l, j\}$ and $pf_2=\{c, g, k\}$.



Definition 6.27:

Under the **read-write tree locking protocol (RWTL)** lock requests and releases must obey LR1 - LR4, WTL1, WTL2, and the two-phase property within each pitfall.

Correctness of RWTL

86

Theorem 6.28:
 $\text{Gen}(\text{RWTL}) \subseteq \text{CSR}.$

TL für beliebigen Einstieg

- TL verlangt nicht zwingend den Einstieg in einen Baum an der Wurzel.
- Jeder beliebige Knoten kann als Einstiegspunkt verwendet werden, allerdings beschränkt sich dann das Sperren auf den so definierten Teilbaum.
- Hieraus kann sich mehr Nebenläufigkeit ergeben.

Freigabe von Sperren

88

Sperren können eher als bei 2-PL freigegeben werden:

Falls alle benötigten Nachfolger von x gesperrt sind,
kann die Sperre von x aufgegeben werden.

Problem:

Wie kann entschieden werden, wann dies der Fall ist ?
(„Was wird **noch** benötigt ?“)

Antworten:

- rein mechanisch: falls alle Nachfolger gesperrt sind
- unter Zusatzkenntnissen: durch Hinweise der Transaktion

Falls nur ein Teil der Nachfolger gesperrt ist, Hinweise aber fehlen, degeneriert RWTL zu S2PL.

Grenzen des Nutzens

Frühe Freigabe von Sperren erhöht die Nebenläufigkeit:
Weniger Sperren werden gehalten, weniger Konflikte, weniger Warten.

Aber:

- Realisierbar nur, wenn der Baum von der Wurzel zu den Blättern durchlaufen wird.
- Zudem liegt die Freigabe des Sperren in der Verantwortung der Transaktion, der Scheduler kennt (zumeist) den Zeitpunkt nicht.
- (Und im Vorgriff:) Die Recovery fordert, dass Sperren bis zum Transaktionsende gehalten werden
⇒ kein Gewinn.

Predicate Locking

(für Interessierte)

Prädikatsperren

Prädikatsperren

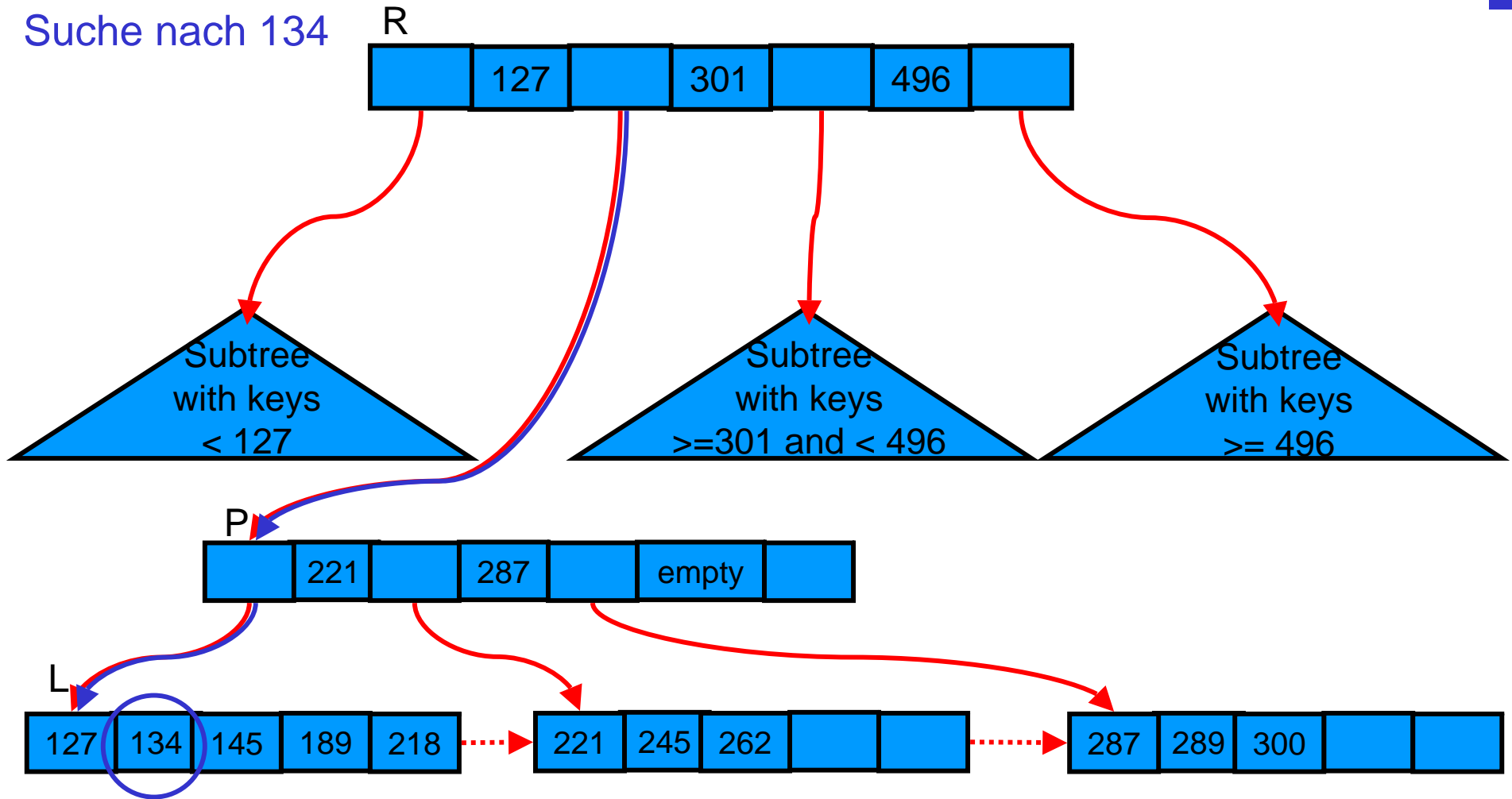
- Prädikatsperren identifizieren die zu sperrenden Datenelemente über eine gemeinsame Eigenschaft (auch: *semantische Sperre*).
 - ◆ Beispiel: Multi-Korngrößen-Sperren mit dem einfachen Prädikat „physisch enthalten“.
- Hier: Schlüsselbereichssperren (key range locks), die über die gesamte Transaktionsdauer gehalten werden. Dieser Bereich kann auch nur ein einzelner Schlüsselwert sein, falls es sich um einen Sekundärindex handelt (mehrere Einträge pro Schlüssel).

B-Baum-Protokolle (1)

- TL nicht zu übernehmen, da Baum in beiden Richtungen durchlaufen werden kann.
- **Notwendigkeit:** Spezielle Protokolle wegen hoher Bedeutung des B-Baum (und Varianten) für Indexstrukturen.
- **Zusatzkenntnisse:** Über B-Baum kennen wir sogar die Algorithmen.
- **Vorgehen:** Die für uns wichtigen Operationen sind `search`, `insert` und `delete`. Da sich `delete` im wesentlichen auf `insert` zurückführen lässt, betrachten wir nur `search` und `insert`.
 - ◆ Basieren auf Schlüsselwerten.

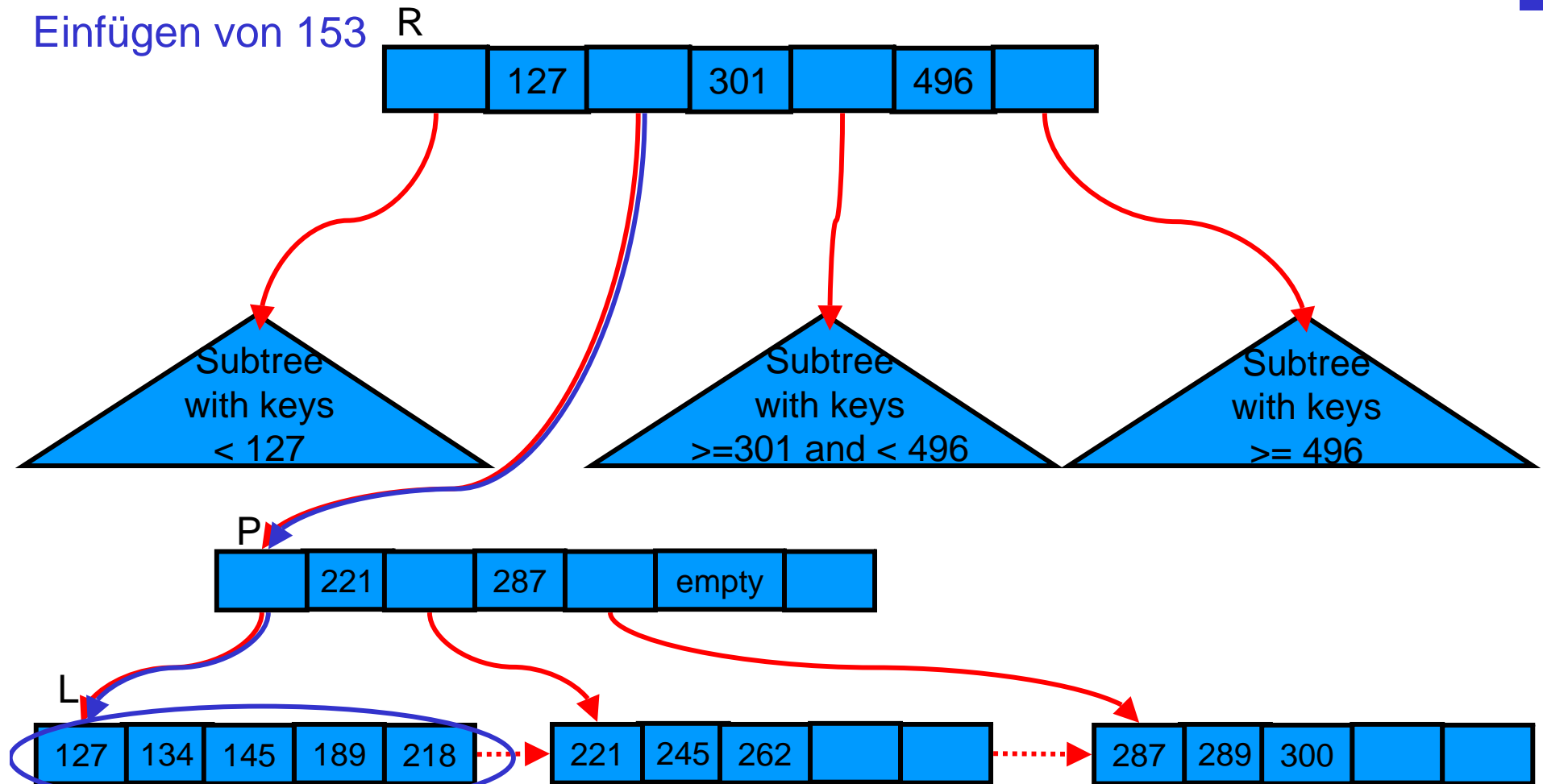
B-Baum-Protokolle (2)

Suche nach 134



B-Baum-Protokolle (3)

Einfügen von 153



Überlauf!

B-Baum-Protokolle (4)

Problem: Eine `insert`-Operation weiß erst bei Erreichen der Blattseite, ob ein Split nötig ist.

Einfache Lösung:

- **TL speziell:** Zunächst Schreibsperrern auf jede zu besuchende Seite zu setzen. Falls diese sich dann bei der anschließenden Inspektion als nicht voll erweist, kann die Sperre beim Vorgänger in eine Lesesperre umgewandelt werden.
 - Dies kann aber zu erheblich weniger Nebenläufigkeit führen.

B-Baum-Protokolle (5)

Problem: Eine `insert`-Operation weiß erst bei Erreichen der Blattseite, ob ein Split nötig ist.

Volles Wissen um und Eingriff in die Algorithmen

- Grundgedanke für das Ändern: Rein lokale Entscheidungen.
- Modifiziere `insert` so, dass es einen Update auf einer Seite durchführen kann, ohne eine Sperre auf dem Vorgängerknoten zu besitzen.
- Also: Entwicklung eines Protokolls, so dass bei `insert` nur w -Sperre auf Seite mit Update gehalten wird, nicht aber Sperre auf dem Vorgängerknoten.
- Falls kein Spalten notwendig ist, kein Problem.

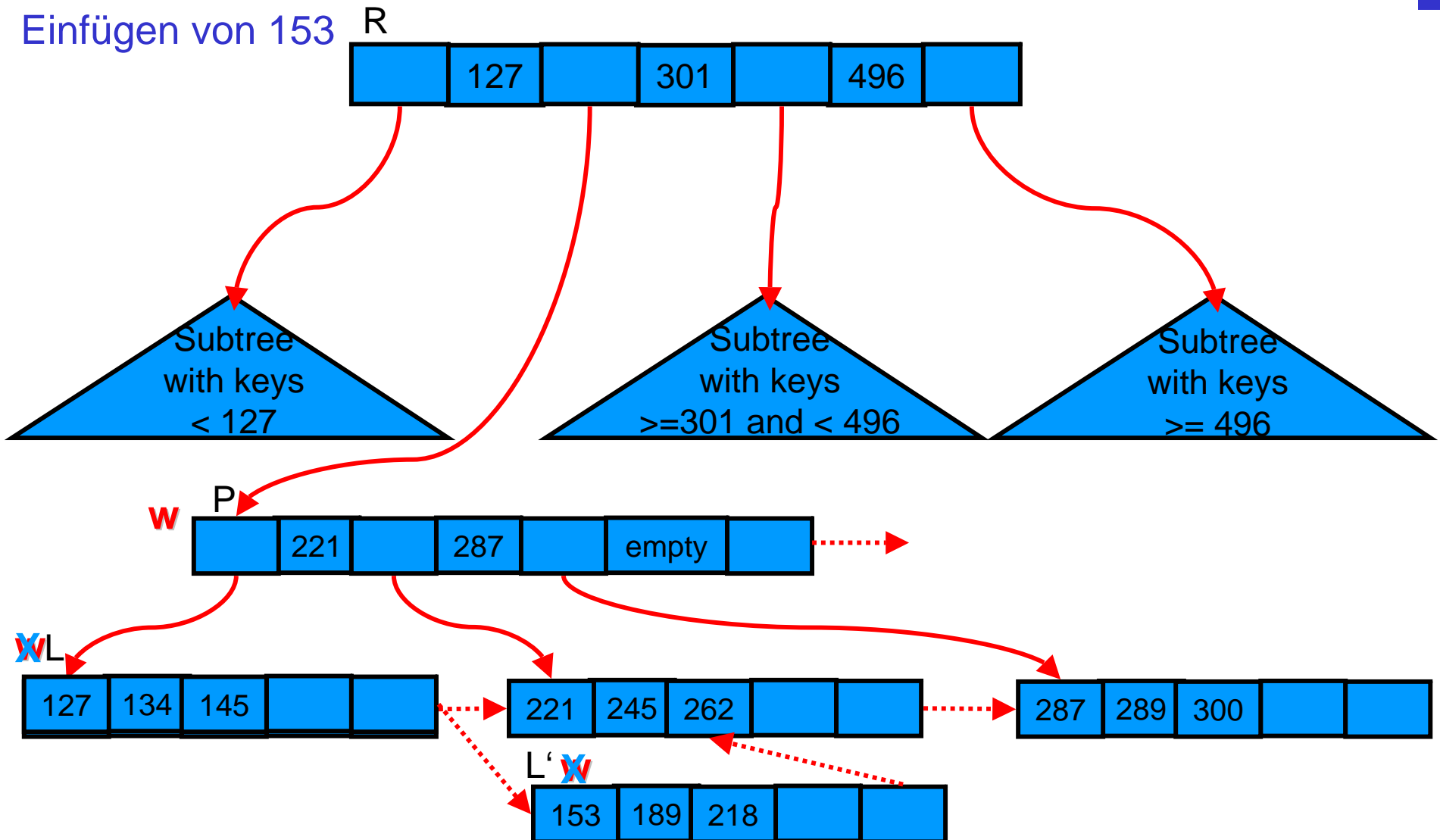
B-Baum-Protokolle (6)

Bei Spalten einer Seite:

- Wie üblich, wird zunächst eine neue Seite L' erzeugt, dann die „obere Hälfte“ der Einträge der vollen Seite L in die neue Seite eingetragen. Die Geschwisterverweise (Links) von L und L' werden aktualisiert. Jetzt kann `insert` die Sperren auf L und L' freigeben. Die Transaktion hält keine weiteren Sperren!
- Im zweiten Schritt wird eine w -Sperrung auf dem Vorgängerknoten angefordert und der Zeiger von L' dort eingetragen.
- Falls der Vorgängerknoten voll ist, erfolgt ein weiteres Spalten.

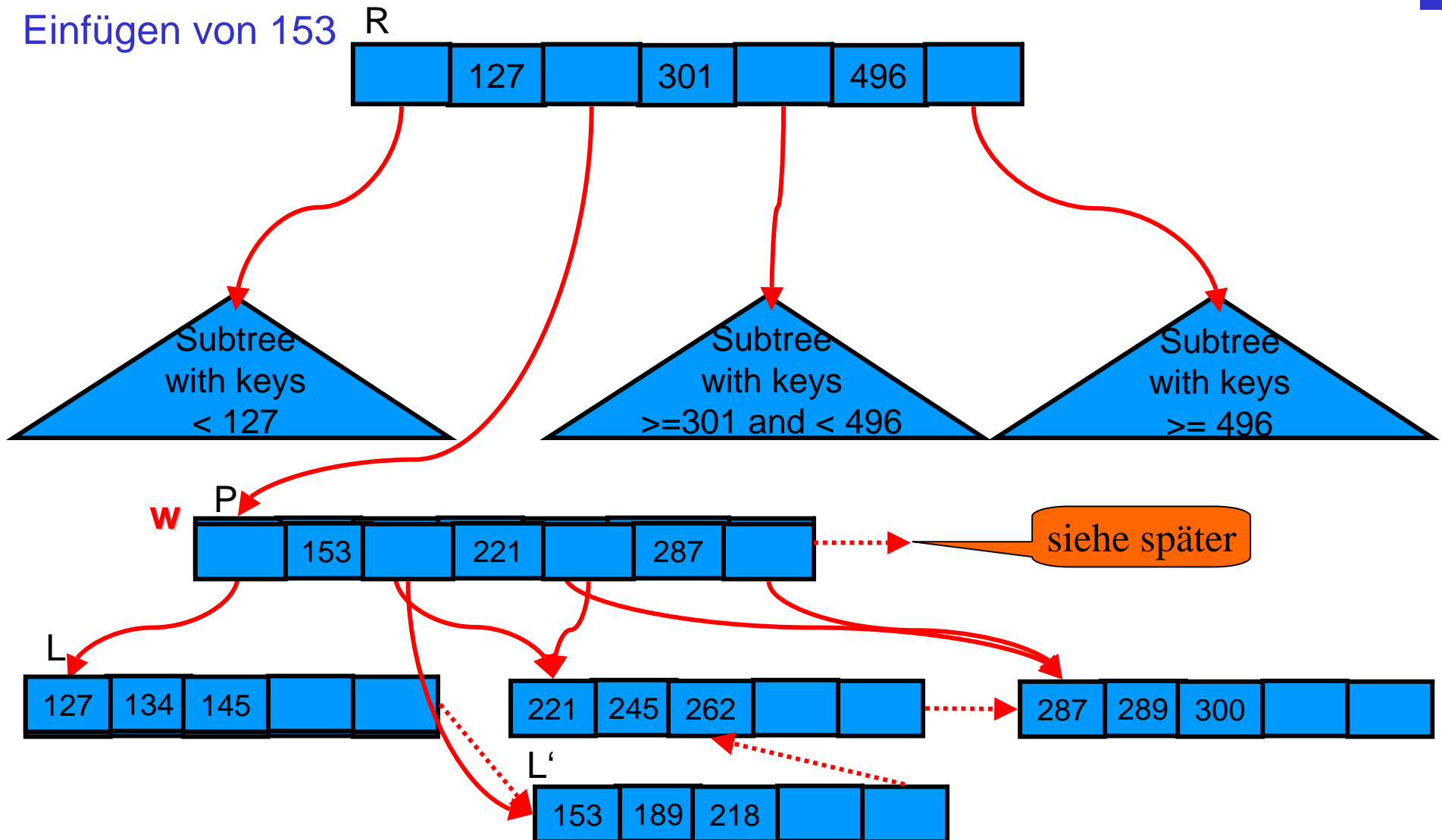
B-Baum-Protokolle (7)

Einfügen von 153



B-Baum-Protokolle (7)

Einfügen von 153



B-Baum-Protokolle (8)

100

- Folge: Beim Lesen muss die Umgebung auf die lokalen Änderungen untersucht werden (lokales Ändern - globales Lesen).
- Die Umgebung kann aber nicht mehr durch Sperren definiert werden, sondern nur durch Inhalte. Diese müssen über zusätzliche Links beschafft werden.
- Folge: Links nunmehr auf allen Ebenen notwendig (**Sperrprotokoll wird in die Baumstruktur eingebaut**)
- Teil 1: Sperr-Protokoll bei `search`: Anfordern einer Lesesperre, direkt nachdem eine Seite gelesen wurde, wird die entsprechende Sperre wieder freigegeben. Danach wird die Sperre der Nachfolgerseite angefordert und diese gelesen.

B-Baum-Protokolle (9)

Teil 2: Systematische Inspektion der Umgebung.

- Es gibt einen Moment, zu dem `insert` keine Sperre hält. Dann kann es von einem `insert` oder `search` überholt werden.
- Da die Seiten von unten nach oben verändert werden, trifft die überholende Transaktion auf eine Änderung, für die es möglicherweise weiter oben noch keinen Hinweis gab, für die also noch kein Nachfolgerzeiger existierte. Dazu muss die Suche so modifiziert werden, dass stets auch die über den Link erreichbare Nachbarseite mit untersucht wird.
- Ebenso gibt es einen Moment, zu dem `search` keine Sperre hält. Überholung durch `search` ist unproblematisch. Überholung durch `insert` wird wie zuvor durch Untersuchung die Nachbarseite behandelt.

B-Baum-Protokolle (10)

Anmerkungen:

- Da beide Operationen, **insert** und **search**, nur dann eine Sperre anfordern, wenn sie keine halten, kann es zu keiner Verklemmung kommen.
- Das Protokoll funktioniert, da es Zusatzwissen über die Algorithmen für **insert**, **search** ausnutzt.
- Die Algorithmen für **insert**, **search** sind nicht mehr unabhängig vom verwendeten Synchronisationsprotokoll.

Hot Spots

Hot Spot: Datenelement, das viele Transaktionen zu ändern wünschen.

→ Systemengpass

→ Gefahr des **Konvoiphänomens:** Bei ungeschickter Schedulingstrategie Ausbildung langer Warteschlangen von Transaktionen, die sich von Hot Spot zu Hot Spot fortpflanzen.

Beispiele für Hot Spots:

- Wurzeln von Bäumen (daher TL-Protokoll!)
- Zähler

Feldzugriffe (1)

104

Strategie: Mischung aus konservativem und aggressivem Scheduling.

Feldzugriff: Eine Aktion (auf einem Hot Spot-Record) wird in zwei Teile zerlegt,

- ein Prädikat und
- eine Transformation.

Betrachte

Prädikat: Bestand ≥ 10
Transformation: Bestand = Bestand - 10;

```
exec sql update hotspot Bestände
set Bestand = Bestand - 10
where Sorte = Weißweine
and Bestand  $\geq 10$ ;
```

Feldzugriffe (2)

Protokoll von Feldzugriffen :

1. Sofortiger Test des Prädikates unter kurzer Lesesperre. (Die Sperre auf das ungeänderte Datenelement wird freigegeben, sobald der Test beendet ist.)
2. Falls der Test zu *falsch* evaluiert, wird abgebrochen.
3. Ansonsten wird ein REDO Log Record mit Prädikat und Transformation angelegt.
4. Bei Erreichen von Commit werden 2 Phasen durchgeführt:
 - ◆ **Phase 1** Alle REDO Log Records der beendenden Transaktion werden bearbeitet, Lesesperren werden angefordert für Feldzugriffe, die keine Transformation beinhalten, Schreibsperren für alle anderen Feldzugriffe. Danach werden alle Prädikate noch einmal evaluiert. Falls mindestens eines zu falsch evaluiert, wird die Transaktion zurückgesetzt. Ansonsten Eintritt in Phase 2 des Commits.
 - ◆ **Phase 2** Alle Transformationen werden angewendet und die Sperren freigegeben.

Feldzugriffe (3)

Vorteil: Statt einer langdauernden Sperre nur noch Kurzzeitsperren.

Nachteile:

- Die Prädikatevaluierung in Phase 1 des Commits kann (wegen zwischenzeitlicher Änderung durch zweite TA) nachträglich doch noch fehlschlagen, wodurch die Transaktion dann zurückgesetzt werden muss.
- Da eigene Änderungen erst am Ende festgeschrieben werden, werden diese durch die Transaktion bei erneutem Lesen nicht berücksichtigt.

Feldzugriffe (4)

Beispiel:

t1 qoh > 150?
qoh := qoh - 150
commit

t2 qoh > 800?
qoh := qoh - 800
commit

t3 qoh > 100?
qoh := qoh - 100
commit

Feldzugriffe (5)

t1	t2	t3	qoh
qoh > 150	qoh > 800 commit qoh > 800 qoh := qoh - 800		1000
commit qoh > 150 qoh := qoh - 150		qoh > 100	200
		commit qoh > 100 F	50

Escrow-Sperren

- **Escrow-Sperre:** Zu beachtendes numerisches Intervall.
- Test ob gegebener Wert *sicher* im Intervall liegt.

t1	t2	t3	Escrow	qoh
			[1000,1000]	1000
qoh>150 qoh:=qoh-150			[850,1000]	
	qoh>800 qoh:=qoh-800 commit		[50,1000] [50,200]	200
commit		qoh>100 F	[50,50]	50

t3 könnte warten, bis fest steht, ob die Untergrenze wieder angehoben wird oder nicht, oder abbrechen.

Non-locking schedulers: Time stamp ordering

Synchronization strategy pursued

111

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. during vs. at commit of a TA

Aggressive scheduler: Set equivalence time to time at the beginning \Rightarrow Try immediate execution of an operation \Rightarrow Little freedom for reordering (fewer equivalent serial schedules can be constructed).

Synchronization by **time stamping**.

Pessimistic strategies
“Watch every step”

Time stamp ordering protocol (1)

Intuition:

- Choose for each transaction as the equivalence time the time of the first operation, i.e., the equivalent serial schedule is practically based on the order in which the transactions were started.
- Conflict equivalence of the real and serial schedules is ensured by checking for each operation whether - without reordering incompatible operations - it could have been executed at the start of the transaction.
- If the check fails abort the transaction.

Time stamp ordering protocol (2)

113

- **Time stamp ordering (TO)** assigns to each transaction t_i a unique time stamp $ts(t_i)$.
 - ◆ Each operation of a transaction is assigned the one and same time stamp of the transaction .
- The *TO* protocol follows the rule :
$$\forall p_i(x) \parallel q_j(x) \in S, i \neq j \quad p_i(x) <_s q_j(x) \Leftrightarrow ts(t_i) < ts(t_j)$$
 - ◆ *TO* scheduler orders – if possible – operations that are in conflict wrt to their time stamps.

Time stamp ordering protocol (3)

114

Theorem 6.29

$Gen(TO) \subseteq CSR$

Sketch of proof:

$t_i \rightarrow t_j \in G(h)$

$\succ \exists p_i(x) \not\parallel q_j(x) \in h \quad p_i(x) <_h q_j(x)$

$\succ ts(t_i) < ts(t_j)$ (interpretation edge!)

Exists cycle $t_1, \dots, t_n, t_1 \in G(h)$.

By induction: $ts(t_1) < ts(t_1)$. Contradiction!

BTO (1)

115

- Base TO (BTO) is a straightforward aggressive implementation of TO . All operations are directly passed on to the data manager (FCFS ordering).
- Late operations are rejected. Operation $p_i(x)$ *is late* if
$$\exists q_j(x) \in s \quad p_i(x) \not\ll q_j(x) \wedge q_j(x) <_s p_i(x) \wedge ts(t_j) > ts(t_i)$$
(Since $q_j(x)$ has already been executed, $p_i(x)$ must be rejected because there is no way to satisfy TO . $\Rightarrow t_i$ must be aborted.
- On restart t_i must be assigned a new and higher time stamp so that one can hope that by now the incompatible operations can correctly be ordered.

BTO (2)

- To determine whether an operation is late, the BTO scheduler holds for each x two values:

max-r-scheduled(x)

- Value of the largest time stamp of earlier read operations on x sent to the data manager.

max-w-scheduled(x)

- Value of the largest time stamp of earlier write operations on x sent to the data manager.

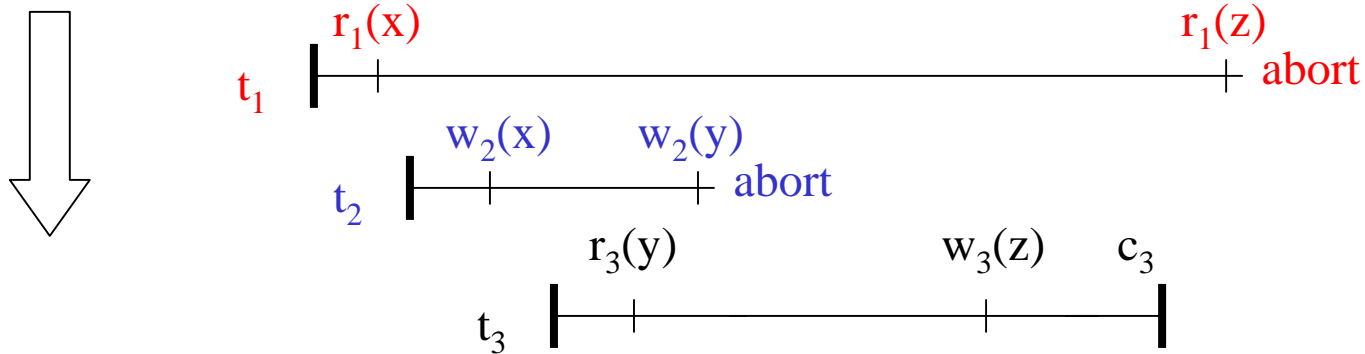
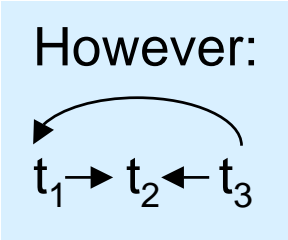
- $p_i(x)$ arrives at the scheduler.
 - ◆ The scheduler compares $ts(t_i)$ with all *max-q-scheduled(x)* where q and p are in conflict.
 - ◆ If $ts(t_i) < \text{max-}q\text{-scheduled}(x)$, $p_i(x)$ is rejected.
 - ◆ Else $p_i(x)$ is passed on to the data manager, and if $ts(t_i) > \text{max-}p\text{-scheduled}(x)$, *max-p-scheduled(x)* is set to $ts(t_i)$.

BTO (3)

Example 6.30

$s = r_1(x) w_2(x) r_3(y) w_2(y) c_2 w_3(z) c_3 r_1(z) c_1$
cannot be generated because :

$s = r_1(x) w_2(x) r_3(y) \cancel{w_2(y)} \cancel{c_2} w_3(z) c_3 \cancel{r_1(z)} \cancel{c_1}$
 a_2
 a_1



$r_1(x) w_2(x) r_3(y) a_2 w_3(z) c_3 a_1$

BTO (4)

Scheduler und Data manager (DM) are autonomous: Coordination needed

- (Aggressive) scheduler and DM must coordinate their activities to ensure that DM maintains the order of the incompatible operations as submitted by the scheduler.
 - ◆ Not needed for 2PL because (conservative) scheduler delays incompatible operations already outside DM.
- Handshake needed: Scheduler counts for each x the r and w operations that were sent to DM but not yet acknowledged as completed:
 r -in-transit(x)
 w -in-transit(x)
- A queue $queue(x)$ maintains operations that could be TO-scheduled but still wait for an acknowledgement by DM of earlier operations that are in conflict.

BTO (5)

Example 6.31

op	action	max-r-scheduled	r-in-transit	max-w-scheduled	w-in-transit	queue
		0	0	0	0	()
$r_1(x)$	to DM	1	1			
$w_2(x)$	wait			2		$(w_2(x))$
$r_4(x)$	wait	4				$(w_2(x), r_4(x))$
$r_3(x)$	wait					$(w_2(x), r_4(x), r_3(x))$
$ack(r_1(x))$	$w_2(x)$ to DM		0		1	$(r_4(x), r_3(x))$
$ack(w_2(x))$	$r_4(x), r_3(x)$ to DM		2		0	()

Striktes TO (1)

120

- Zur Erinnerung strikte Historien: Schreibt eine Transaktion t_i das Datenelement x , so darf eine weitere Transaktion t_j , die erst anschließend auf x zugreift, dies erst nach dem Ende von t_i tun.
- Der strikte TO-Scheduler arbeitet wie der Basis-TO-Scheduler, außer dass $w\text{-in-transit}(x)$ erst bei a_j oder c_j auf 0 gesetzt wird.
- Hierdurch werden $r_j(x)$ und $w_j(x)$ mit $ts(t_j) > ts(t_i)$ verzögert, bis t_i beendet ist. $w\text{-in-transit}(x)$ verhält sich wie eine Sperre.
- Trotzdem: Auch jetzt können Verklemmungen nicht auftreten, da t_j nur auf t_i wartet, falls $ts(t_j) > ts(t_i)$.

Striktes TO (2)

121

Beispiel:

$$h = r_2(x) w_3(x) c_3 w_1(y) c_1 r_2(y) w_2(z) c_2$$

h ist CSR: $G(h): t_1 \rightarrow t_2 \rightarrow t_3$

h ist strikt: $w_1(y) <_h r_2(y)$ und $c_1 <_h r_2(y)$.

Falls $ts(t_1) < ts(t_2) < ts(t_3)$, ist h strikt TO.

SG-basierte Protokolle

Idee: Scheduler basiert auf Konfliktgraphtests.

- Ein SGT-Scheduler baut einen **Serialisierbarkeitsgraph** (SG) des Schedule explizit auf. Operationen ändern den SG. Anpassungen gegenüber Konfliktgraph:
 - ◆ Der SG enthält neben abgeschlossenen TAs auch aktive. (Laufende Ermittlung erforderlich!)
 - ◆ Der SG enthält nicht jede abgeschlossene TA. (Der größte Teil der Vergangenheit interessiert nicht!)
- Der SGT-Scheduler achtet darauf, dass der SG immer zyklensfrei bleibt.

Basis-SGT (1)

123

Wenn der SGT-Scheduler eine Operation $p_i(x)$ erhält,

1. falls $t_j \notin \text{SG}$: einfügen
 2. für alle $q_j(x) <_s p_i(x)$, $i \neq j$, $q_j(x) \parallel p_i(x)$: $\text{SG} \cup \{t_j \rightarrow t_i\}$
 3. falls SG jetzt Zyklus enthält: abort t_i . Nach Bestätigung von Daten-Verwalter: $\text{SG} \setminus \{t_j \rightarrow t_i\}$
 4. sonst: plane $p_i(x)$ ein. Sobald alle in Konflikt stehenden Operationen bestätigt, Weitergabe an Daten-Verwalter.
- Zur Bestimmung von Bedingung 2 zusätzlich für jede t_j
 r -scheduled(t_j) und w -scheduled(t_j)
in denen die Lese- und Schreibmengen (Datenelemente) von t_j gehalten werden (SG zu grobe Abstraktion!).
 - Zur Bestimmung von Bedingung 4 wieder q -in-transit(x).
 - Um Striktheit zu gewährleisten, geht man analog zu striktem TO vor.

Basis-SGT (2)

Wann können TA's aus SG gelöscht werden?

- Nicht direkt nach commit! Betrachte

$$h = r_{k+1}(x) w_1(x) w_1(y_1) c_1 w_2(x) w_2(y_2) c_2 \dots w_k(x) w_k(y_k) c_k$$

$$\text{SG: } t_{k+1} \rightarrow t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_k$$

- Jetzt erreiche $w_{k+1}(z)$ den Scheduler.
- Dieser muss testen, ob $z \in \{x, y_1, \dots, y_k\}$ (falls ja: Zyklus!).
- Hierzu wird jedoch genau die Menge $\{t_1, \dots, t_k\}$ benötigt, obwohl diese Transaktionen schon abgeschlossen sind.

Basis-SGT (3)

Triviale Feststellung:

Abgeschlossene Transaktionen können aus SG entfernt werden, wenn sie nicht mehr an einem Zyklus teilnehmen können.

Lösung:

- Für Zyklus benötigen Knoten mindestens eine einfallende und eine ausgehende Kante.
- Abgeschlossene Transaktionen ohne einfallende Kanten können aus SG entfernt werden, da nach einem Commit nur ausgehende Kanten entstehen können.

Optimistic schedulers

Synchronization strategy pursued

127

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. during vs. at commit of a TA

Bei Rücksetzen muss die gesamte Transaktion unschädlich gemacht werden.
Anwendbarkeit: Bei geringer Konflikthäufigkeit

That just leaves equivalence time to be set to commit time \Rightarrow
This time known only at the end \Rightarrow Failure possible \Rightarrow Limited opportunities for reordering.

Optimistic strategies
“Go ahead and then see”

Optimistische Verfahren (1)

Drei Transaktionsphasen:

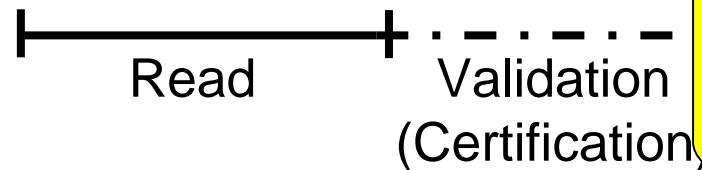


1. **Read-Phase:** Die Transaktion wird ausgeführt, aber alle *w*-Aktionen wirken nur auf lokalen Arbeitsbereich. (Keine Verzögerungen anderer Transaktionen!)
 2. **Validation-Phase:** Eine commit-willige TA wird validiert, d.h. es wird geprüft, ob sie *CSR* abgelaufen ist.
 3. **Write-Phase:** Ist das Ergebnis der Validierung positiv, wird der Inhalt des Arbeitsbereichs in die Datenbasis übertragen.
2. und 3. werden ununterbrechbar ausgeführt („**Valwrite-Phase**“).

Optimistische Verfahren (1)

129

Drei Transaktionsphasen:



Hört sich harmlos an, ist aber implementierungstechnisch nicht ganz unproblematisch. Realisierung z.B. durch Sperren während dieser Phase.

1. **Read-Phase:** Die Transaktion wird ausgeführt, aber die W-Aktionen wirken nur auf lokalen Arbeitsbereich. (Keine Verzögerungen anderer Transaktionen!)
 2. **Validation-Phase:** Eine commit-willige TA wird validiert, d.h. es wird geprüft, ob sie *CSR* abgelaufen ist.
 3. **Write-Phase:** Ist das Ergebnis der Validierung positiv, wird der Inhalt des Arbeitsbereichs in die Datenbasis übertragen.
2. und 3. werden *ununterbrechbar* ausgeführt („**Valwrite-Phase**“).

Optimistische Verfahren (2)

130

Satz 6.32

Sei G ein DAG. Wenn ein neuer Knoten ohne ausgehende Kanten zu G hinzugefügt wird, so ist auch der resultierende Graph azyklisch.

Wir werden den Satz wie folgt nutzen: Die Validierungsprotokolle werden überprüfen, ob die zu validierende Transaktion einen azyklischen Konfliktgraphen azyklisch belässt \Rightarrow Transaktion darf nur einen Knoten ohne ausgehende Kanten hinzufügen.

Definition 6.33 (Read-Set/Write-Set)

- Das **Read-Set** $RS(t)$ ist die Menge aller von t gelesenen Datenelemente.
- Das **Write-Set** $WS(t)$ ist die Menge aller von t geschriebenen Datenelemente.

Optimistische Verfahren (3)

Zwei Alternativen:

- Rückwärtsvalidierung (backward-oriented optimistic concurrency control, BOCC)
 - ◆ zu validierende Transaktion wird gegen die schon abgeschlossenen Transaktionen getestet
- Vorwärtsvalidierung (forward-oriented optimistic concurrency control, FOCC)
 - ◆ zu validierende Transaktion wird gegen die gleichzeitig ablaufenden Transaktionen, die sich noch in der Lese phase befinden, getestet.

BOCC (1)

132

BOCC validation of t_j :

- Compare t_j to all previously committed t_i
- Accept t_j if one of the following holds
 - ◆ t_i has ended before t_j has started, or
 - ◆ $RS(t_j) \cap WS(t_i) = \emptyset$

Theorem 6.34:

$Gen(BOCC) \subset CSR$

Proof:

- Assume that $G(CP(s))$ is acyclic.
- The new node is last in the serialization order, and has no outgoing edges.

BOCC (2)

133

Diese Transaktionen wurden bereits validiert, das heißt, bis zu diesem Zeitpunkt ist G azyklisch.

BOCC validation of t_j :

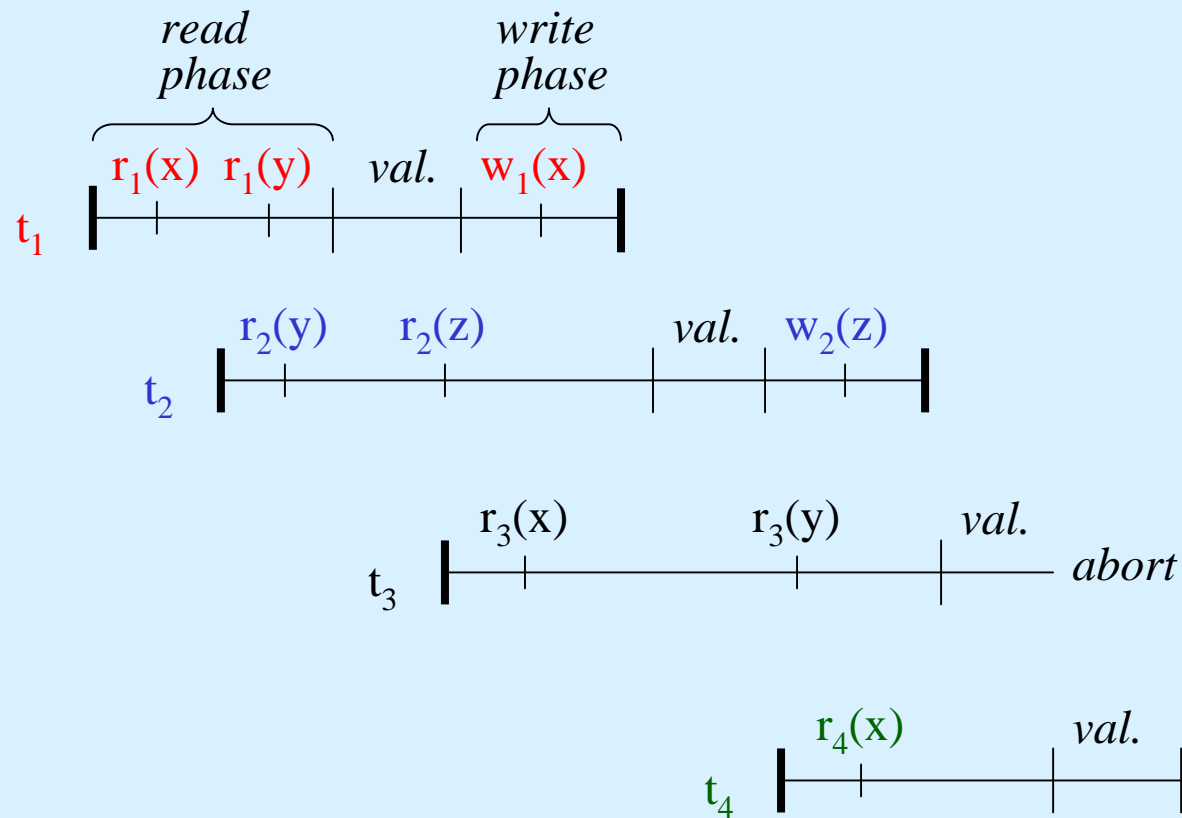
- Compare t_j to all previously committed t_i
 - Accept t_j if one of the following holds
 - ◆ t_i has ended before t_j has started, or
- $$RS(t_j) \cap WS(t_i) = \emptyset$$

Falls Konflikt zwischen t_i und t_j existiert, gibt es die Kante (t_i, t_j) . Die umgekehrte Kante kann es nicht geben.

t_j hatte keine Gelegenheit von t_i zu lesen. Es kann also wieder keine Kante (t_j, t_i) geben.

BOCC (3)

Example 6.35



FOCC (1)

FOCC validation of t_j :

- Compare t_j to all concurrently active t_i
 - ◆ (which must be in their read phase)
- Accept t_j if $WS(t_j) \cap RS^*(t_i) = \emptyset$
 - ◆ where $RS^*(t_i)$ is the current read set of t_i

Theorem 6.36:

$Gen(FOCC) \subset CSR$

Proof:

- Assume that $G(CP(s))$ is acyclic.
- Adding a newly validated transaction can only insert edges going into the new node, but no outgoing edges because for all previously committed t_k the following holds:
 - If t_k committed before t_j started, then no edge (t_j, t_k) is possible.
 - If t_j was in its read phase while t_k validated, then $WS(t_k)$ must be disjoint with $RS^*(t_j)$ and all later reads of t_j and all writes of t_j must follow t_k , so no edge (t_j, t_k) is possible.

FOCC (2)

FOCC validation of t_j :

- Compare t_j to all concurrently active t_i
 - ◆ (which must be in their read phase)
- Accept t_j if $WS(t_j) \cap RS^*(t_i) = \emptyset$
 - ◆ where $RS^*(t_i)$ is the current read set of t_i

Remarks:

- FOCC is much more flexible than BOCC: Upon unsuccessful validation of t_j it has three options:
 - ◆ abort t_j
 - ◆ abort one of the active t_i for which $RS^*(t_i)$ and $WS(t_j)$ intersect
 - ◆ wait and retry the validation of t_j later (after the commit of the intersecting t_i)
- Read-only transactions do not need to validate at all.

FOCC (3)

Example 6.37

