Database Management Systems II

Robert Gottstein

gottstein@dvs.tu-darmstadt.de

Exercise 7.1

Data Contention, Resource Contention and Thrashing

Exercise 7.1
Data Contention, Resource Contention and Thrashing

a) Explain the difference between **data contention and resource contention** and the way they could lead to data thrashing.

As workload (number of transactions) increases, transaction throughput increases up to a point and then drops.

This phenomenon is called **Thrashing**.

Thrashing is caused by contention for data/resources, which results in long waiting queues.

Exercise 7.1
Data Contention, Resource Contention and Thrashing

TECHNISCHE UNIVERSITÄT DARMSTADT

We distinguish two types of contention:

1.Resource Contention (RC)

refers to contention for resources such as main memory, CPU time, I/O channels etc.

E.g. Processes may contend for main memory, causing frequent page faults. The system ends up spending most of the time in swapping pages in and out.

Exercise 7.1
Data Contention, Resource Contention and Thrashing

2. Data Contention (DC)

- refers **to contention for data access** (contention for **locks**).
- DC leads to **waiting queues** caused by lock-requests.
- Once the thrashing point is reached, each **new transaction usually results in multiple blocked transactions**.
- DC Thrashing can be **caused by blocking or by deadlocks** (followed by restarts).
- Experiments show that DC-thrashing is **actually caused by blocking and not by restarts**.
  Deadlocks are rare before the thrashing point and dominate after it.

Exercise 7.1
Data Contention, Resource Contention and Thrashing

b) What could be done to reduce data contention and achieve throughput beyond the thrashing point?

To reduce data contention:

configure the ***blocking timeout interval***

use a finer ***locking granularity***

decrease ***transaction isolation levels***

apply ***transaction chopping***

Exercise 7.1
Data Contention, Resource Contention and Thrashing

TECHNISCHE UNIVERSITÄT DARMSTADT

Conflicts can be resolved by **blocking** or **restarting**:

- **A pure blocking policy** is selfish. Transaction's (own) work is preserved at the cost of locks being held without being used.

- **A pure restart policy** is self-sacrificing. Transaction is aborted (losing its work), so that locks can be released and other transactions not hindered, while it waits.

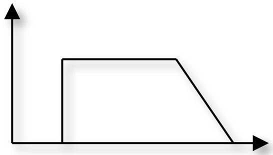
Exercise 7.1
Data Contention, Resource Contention and Thrashing

If ***transactions abort quickly and ResCon is low***

→ *restart policy has a throughput slightly lower* than that of the blocking policy before the latter's DC thrashing point.

But surprisingly**, *after the thrashing point for blocking, the restart policy has a higher throughput*.**

Another way to bring throughput beyond the limit imposed by blocking through DC-thrashing, is to use **Conservative 2PL.**



Exercise 7.2

TECHNISCHE UNIVERSITÄT DARMSTADT

Tree Locking (TL) Protocols

Exercise 7.2
Tree Locking (TL) Protocols

Explain briefly the main principles of TL protocols

Data items are

structured as nodes of a tree and

transactions always access data items by following paths in the tree.

➡ **Relax the two-phase rule** of 2PL and develop protocols that:

allow more concurrency

still ensuring serializability.

Exercise 7.2
Tree Locking (TL) Protocols

TL Rules:

1. Conflicting operations are scheduled and processed in the order in which corresponding locks are obtained (rules 1 and 2 of 2PL)
2. To set a lock on **'x'** (where **'x'** is not the root), T_i must have a lock on its parent.
3. Once a T_i releases a lock on **'x'** it may not subsequently request a lock on **'x'**.

Lock Coupling Principle:

Locks are obtained in *root-to-leaf* order. The scheduler can release $ol_i[x]$ only after it has obtained the locks T_i needs on x's children.

Exercise 7.2
Tree Locking (TL) Protocols

Rules 1) to 3) are not enough to ensure serializability!

Theorem:

A TL scheduler produces serializable executions provided that every transaction sets either only read locks or only write locks.

Note: We assume that leaf nodes are not linked!

Exercise 7.2
Tree Locking (TL) Protocols

b) Explain how TL protocols can be used to synchronize access to B+-Trees.

Exercise 7.2
Tree Locking (TL) Protocols

Two types of transactions: *Read* and *Update*.

To ensure serializability:
- enforce the TL rules
- read transactions set only read locks
- update transactions set only write locks

Problem:

When to release locks on tree nodes? The earlier the better, otherwise the advantage of TL is lost.
Read locks can be released as soon the lock on child node is obtained.
For **write locks** we need to be careful since updates might require splits/merges all the way up to the root.

Exercise 7.2
Tree Locking (TL) Protocols

Solution:

Bayer and Schkolnik Algorithm

Identify safe nodes that can't produce overflows or underflows.
Release locks on parents when safe node is reached.

Exercise 7.2
Tree Locking (TL) Protocols

Safe Node:

Node such that changes will not propagate up beyond this node.

Safe for Insert:

Node is not full.

Safe for Delete:

If we remove one entry, node is still at least half full.

15	25	37	68

Safe for delete, unsafe for insert

15	25		

Safe for insert, unsafe for delete

Exercise 7.2
Tree Locking (TL) Protocols

There exist **2 types of locks:**
read-locks **r**, exclusive locks ξ

Read lock:

lock *root* **with** *r lock*
current = *root*
while *current* != *leaf* **do**
begin
    lock *son* **with** *r lock*
    release *r* **on** *current*
    current = *locked son*
end

Exercise 7.2
Tree Locking (TL) Protocols

Update lock:

lock *root* **with** ξ *lock*

current = *root*

while *current* != *leaf* **do**

begin

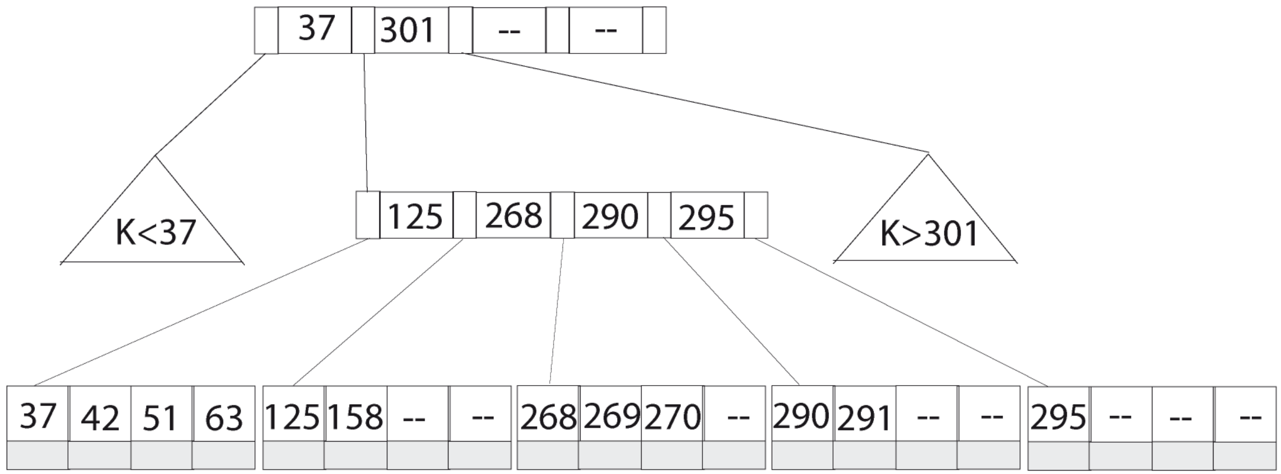
lock *son* **with** ξ *lock*
current = *locked son*
if *current is safe* **then**
release *lock* **on** *parents*

end

Why?

Exercise 7.2
Tree Locking (TL) Protocols

c) Given is the following Tree:



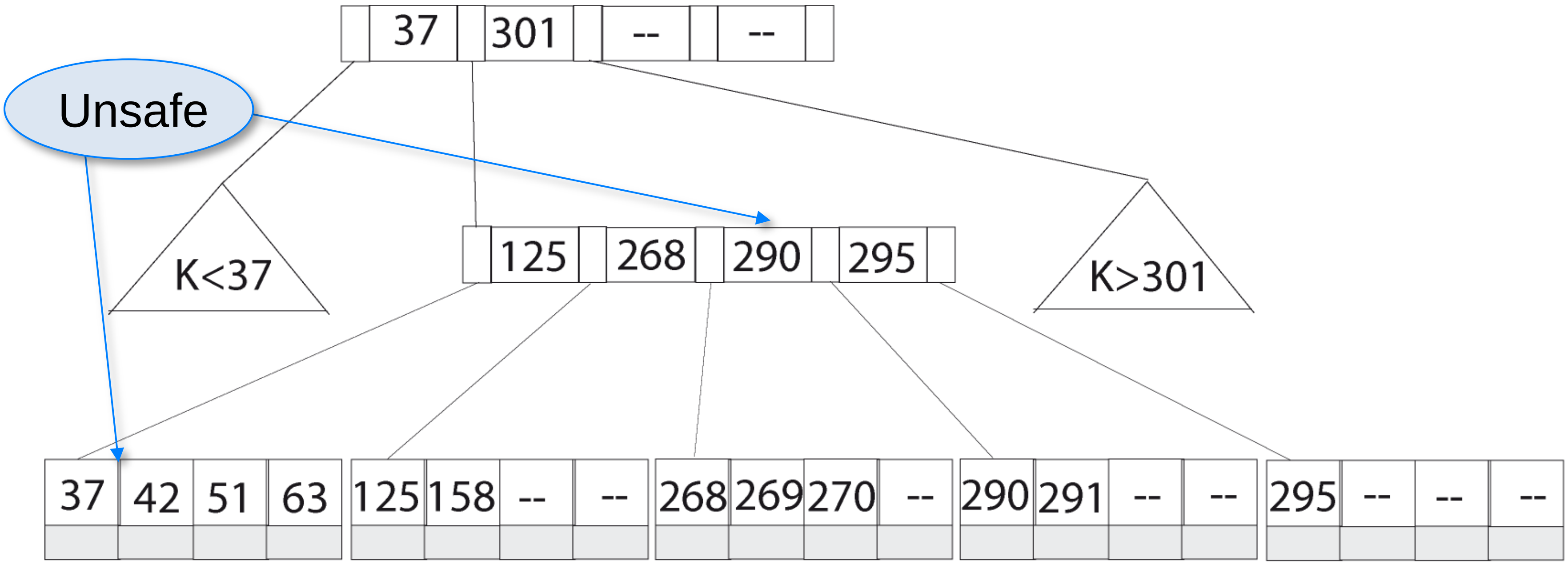
Which nodes are **unsafe** (safe) **for the insert/delete** operation?

Execute **Insert(271)** and **Insert(272)**(Bayer/Schkolnick)

To what extent are exclusive locks unnecessarily set in the Bayer/Schkolnick algorithm? How can this be avoided?

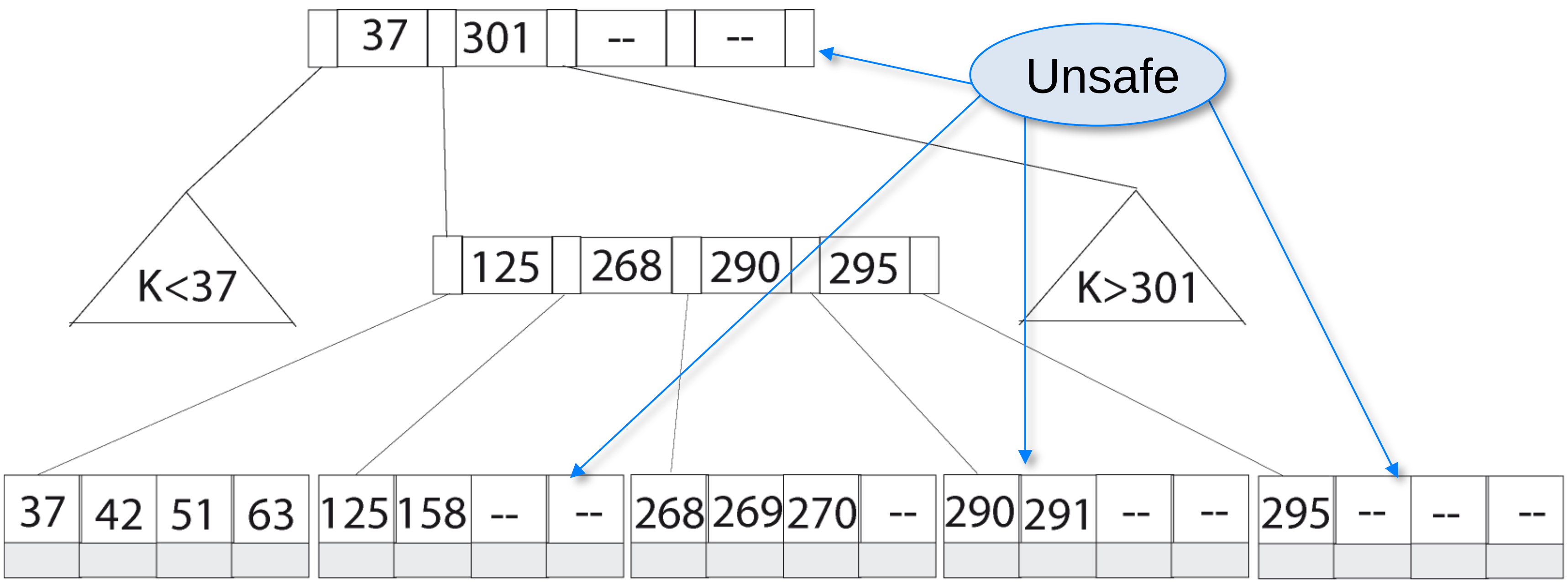
Exercise 7.2
Tree Locking (TL) Protocols

Which nodes are **unsafe** (safe) **for the insert** operation?



Exercise 7.2
Tree Locking (TL) Protocols

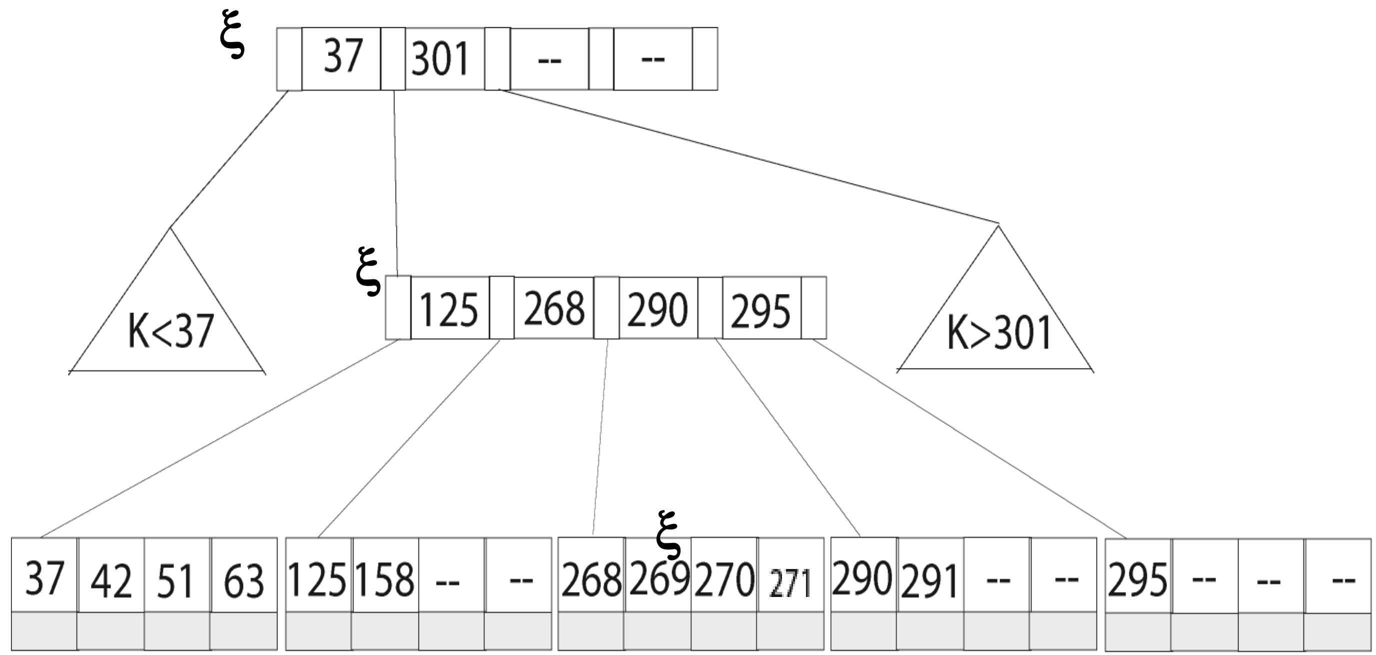
Which nodes are **unsafe** (safe) **for the delete** operation?



Exercise 7.2
Tree Locking (TL) Protocols

- Execute **Insert(271)** and **Insert(272)** (Bayer/Schkolnick)

Insert 272:



Exercise 7.2
Tree Locking (TL) Protocols

Insert(271):

Some locks were set unnecessarily.

Insert(272):

All set locks were necessary.

The problem is that *until we reach the leaf node* we *can't determine* whether further ξ*-locks will be needed.*

Exercise 7.2
Tree Locking (TL) Protocols

How to avoid this?

1) **On update request r locks***.* Set ξ lock on leaf only. If leaf full, release all locks and ***repeat*** using the Bayer/Schkolnik algorithm. (Probing)

2) **On update request r locks**, releasing **locks on ancestors as safe nodes are reached**. Set ξ lock on leaf only. If leaf full, upgrade locks to ξ up to first safe node. **Problem - Deadlocks**! (Due to lock upgrades)

3) **On update request *might-write (MW) locks* (conflict with MW and ξ, but not with r)**, releasing locks on ancestors when a safe node is reached. If leaf full, convert all MW locks to ξ locks top-down starting from node nearest to root. Deadlocks due to lock upgrades are eliminated. (Due to blocking of MW)

Exercise 7.2
Tree Locking (TL) Protocols

Some Final Remarks:

(2) and (3) do ***not necessarily ensure complete serializability*** (transactions can overtake each other – one traverses faster).

Further, if we relax the restriction of not linking leaf nodes, this would hold for (1) as well (more than one path from root to leaf nodes).

Nevertheless, correctness in this specific case is not compromised. The actual data is stored in the leaves. Inner nodes only direct us to the data. If we make sure that leaves are locked according to 2PL, correctness will be ensured.

Exercise 7.3

TECHNISCHE UNIVERSITÄT DARMSTADT

Timestamp Ordering Protocols

Exercise 7.3
Timestamp Ordering Protocols

a) Explain why the TO Rule guarantees serializability.

Each transaction is assigned a **unique timestamp** when started. Timestamps are generated in increasing order.

TO-Rule:

If $o_m[x]$ and $p_n[x]$ are conflicting operations, then the DM processes $o_m[x]$ before $p_n[x]$ iff $ts(T_m) < ts(T_n)$.

In other words, conflicting operations are executed in the order of their transaction's timestamps.

If H is a TO-produced history, a cycle $T_m \rightarrow \ldots \rightarrow T_m$ in SG(H) would imply that $ts(T_m) < ts(T_m)$. Since this is impossible, no cycles exist and H is SR.

Exercise 7.3
Timestamp Ordering Protocols

b) Describe a possible implementation of a TO-based Scheduler.
What data structures are used?
How are the ***Basic TO Rules*** enforced?

Basic TO:

An aggressive TO variant:

Operations are passed on immediately.

Operations that arrive late are rejected ==> restart transaction with a new timestamp.

Exercise 7.3
Timestamp Ordering Protocols

Data Structure:

set_of_data_items	**DB**;
timestamp	**max_r_scheduled**[DB];
timestamp	**max_w_scheduled**[DB];
int	**r_in_transit**[DB];
int	**w_in_transit**[DB];
schedule_queue	**queue**[DB];

Operation om[x] is "too late" if:

For o=r : ts(Tm) < max_w_scheduled[x]

For o=w : (ts(Tm) < max_w_scheduled[x]) $\vee$
(ts(Tm) < max_r_scheduled[x])

Exercise 7.3
Timestamp Ordering Protocols

r_m[x]:

```
if (ts(T_m) < max_w_scheduled[x]) {
    reject it and abort T_m;
} else {
    if ((w_in_transit[x] = 0) AND
        (no write operations are waiting in queue[x])) {
        schedule(r_m[x]);
        r_in_transit[x]++;
        max_r_scheduled[x] = max(ts(T_m), max_r_scheduled[x]);
    } else {
        add r_m[x] to queue[x] making sure that conflicting
        operations are ordered in timestamp order
    }
}
```

Exercise 7.3
Timestamp Ordering Protocols

$w_m[x]$:

if ($ts(T_m)$ < max_r_scheduled[x] OR $ts(T_m)$ < max_w_scheduled[x]) {
    reject it and abort T_m;
} **else** {
    if (r_in_transit[x] = 0 **AND** w_in_transit[x] = 0) {
        schedule(w_m[x]);
        w_in_transit[x]++;
        max_w_scheduled[x] = $ts(T_m)$;
    } **else** {
        add w_m[x] to queue[x] making sure that conflicting operations are ordered in timestamp order
    }
}

Exercise 7.3
Timestamp Ordering Protocols

If $o_m[x]$ too late, then reject it.

Else add it to queue[x], making sure that conflicting operations are queued in timestamp order.

A handshake is needed to ensure operations are processed in the order in which they are scheduled.

Exercise 7.3
Timestamp Ordering Protocols

c) Given are the transactions T_1 (*timestamp=1*), T_2 (*timestamp=5*) and T_3 (*timestamp=10*).
T_1, T_2 and T_3 send the following operations to the scheduler:

Point in Time	T1	T2	T3
1	BOT		
5		BOT	
10	w1[x]		BOT
11		r2[x]	
12			r3[x]
13		w2[x]	
14		c2	
15			w3[x]
16	c1		
17			c3

Exercise 7.3
Timestamp Ordering Protocols

We assume that:

if an operation is sent to the Data Manager at point in time *t* => the scheduler *receives confirmation* from the Data Manager at point in time *t+1 for* ***read operations*** *and t+2 for* ***write operations****.*

Parallel read operations are possible.

Show how the **TO data structures are used**.

Time	BTO Test	w-max-sched[x]	r-max-sched[x]	w-in-transit[x]	r-in-transit[x]	queue[x]
...						

If a transaction T sets o_max_scheduled[x] to TS(T) and is subsequently aborted can o_max_scheduled[x] be restored to its before image with respect to T?

Exercise 7.3
Timestamp Ordering Protocols

TECHNISCHE UNIVERSITÄT DARMSTADT

T	BTO Test	w-max-sched[x]	r-max-sched[x]	w-in-transit [x]	r-in-transit [x]	Queue [x]
9		0	0	0	0	{}
10	0<=1; 0<=1	1	0	1	0	{}
11	1<=5	1	0	1	0	r2
12	1<=10	1	10	0	2	{}
13	1<=5; ! 10<=5	1	10	0	0	{}
14	a2	1	10	0	0	{}
15	1<=10; 10<=10	10	10	1	0	{}
16	c1	10	10	1	0	{}
17	c3	10	10	0	0	{}

Point in Time	T1	T2	T3
1	BOT		
5		BOT	
10	w1[x]		BOT
11		r2[x]	
12			r3[x]
13		w2[x]	
14		a2	
15			w3[x]
16	c1		
17			c3

Abort T2! Since T3‘s timestamp increased r-max-sched

flashyDB DVS

Exercise 7.3
Timestamp Ordering Protocols

Note:

If a transaction T sets o_max_scheduled[x] to TS(T) and is subsequently aborted can o_max_scheduled[x] be restored to its before image with respect to T?

If no other operations have been executed on x since T set o_max_scheduled[x] $\rightarrow$ Yes.

Tradeoff

overhead in maintaining before-images vs. reducing the chance of aborts caused by lately sent operations on x.

Exercise 7.3
Timestamp Ordering Protocols

d) Discuss the following issues in the context of the Basic TO Method:

- Deadlocks
- Livelocks, Starvation
- Strictness

Exercise 7.3
Timestamp Ordering Protocols

- **Deadlocks** cannot occur, since transactions always wait for transactions with lower timestamps → a cycle in the WFG cannot exist.

- **Livelock** is possible - when a transaction is repeatedly restarted, because it sends an operation too late. This can occur if a transaction sends an operation on a heavily accessed data item some time after the transaction starts executing.

Exercise 7.3
Timestamp Ordering Protocols

***Improvement of BTO*:**

If **TS(o) > (max_o_scheduled[x] + Δ) then**
delay op for a while before scheduling it.

Thus, there is a better chance that any conflicting operations with smaller timestamps will arrive in time to be scheduled.

Strictness:

BTO-produced histories are not always strict.

To enforce strictness we could delay resetting w_in_transit[x] to 0 ***until transactions' commit or abort is acknowledged***.

Optimistic Concurrency Control

Exercise 7.4
Optimistic Concurrency Control

a) Explain the different paradigms of pessimistic and optimistic CC methods.

Pessimistic Methods:
Assume that every operation is potentially conflicting and therefore check for conflicts at each operation request.

Optimistic Methods:
Assume that conflicts are rare and execute operations as they arrive without checking for conflicts.

Aggressive vs. Conservative Schedulers:
avoid delaying operations, risking to reject them later
avoid rejecting operations, by delaying them

Exercise 7.4
Optimistic Concurrency Control

TECHNISCHE UNIVERSITÄT DARMSTADT

b) Describe the typical phases in the execution of transactions by an optimistic CC scheduler.

1. **Execution Phase (also called Read Phase)**
   Transaction's operations are executed. Writes are stored in local copies, invisible to other transactions.
2. **Validation Phase**
   The CC criterion is checked (e.g. Locking, TO)
3. **If Validation Passed: Write Phase**
   If a transaction passes validation its updates are written to the DB and become visible to other transactions.

Exercise 7.4
Optimistic Concurrency Control

TECHNISCHE UNIVERSITÄT DARMSTADT

c) Describe a possible implementation of an optimistic CC protocol.

- What criteria are used for validation?
- Which of these criteria for backwards validation would be sufficient, if additional information about the relative order of transaction's phases is available?
  When should transaction T_n be backwards validated with respect to T_m, if:
  1. end_write_phase(T_m) < start_read_phase(T_n)
  2. end_write_phase(T_m) < start_write_phase(T_n)
  3. end_read_phase(T_m) < end_read_phase(T_n)
- Which transactions must the scheduler backwards-validate against?
  When can the protocol data (read/write sets) for a transaction be deleted?

Exercise 7.4
Optimistic Concurrency Control

What criteria are used for validation?

Let **N(T)** be the validation timestamp of T
$\mathbf{T_m}$ is validated before $\mathbf{T_n} \Leftrightarrow \mathbf{N(T_m) < N(T_n)}$

Idea:
If $\mathbf{N(T_m) < N(T_n)}$, then $\mathbf{T_m}$ should be before $\mathbf{T_n}$ in an equivalent serial history.

To enforce CSR:
Make sure all conflicting operations are executed in the order of the respective transaction's timestamps:
i.e. if $\mathbf{N(T_m) < N(T_n)}$, then all conflicting operations of $\mathbf{T_m}$ and $\mathbf{T_n}$ must be executed in the order: first $\mathbf{T_m}$ then $\mathbf{T_n}$.

Exercise 7.4
Optimistic Concurrency Control

- Which of these criteria for *backwards validation* would be sufficient, if additional information about the ***relative order of transaction's phases*** is available?

Transaction $\mathbf{T_n}$ is validated with OK, if for every transaction $\mathbf{T_m}$ *validated so far*, every pair of conflicting operations of $\mathbf{T_n}$ and $\mathbf{T_m}$ are executed in the order: first $\mathbf{T_m}$ then $\mathbf{T_n}$.

Exercise 7.4
Optimistic Concurrency Control

When should transaction T_n be backwards validated with respect to T_m, if:

1. end_write_phase(T_m) < start_read_phase(T_n)
2. end_write_phase(T_m) < start_write_phase(T_n)
3. end_read_phase(T_m) < end_read_phase(T_n)

Exercise 7.4
Optimistic Concurrency Control

TECHNISCHE UNIVERSITÄT DARMSTADT

We can use the following **criteria for excluding** existence of a potential **conflict**:

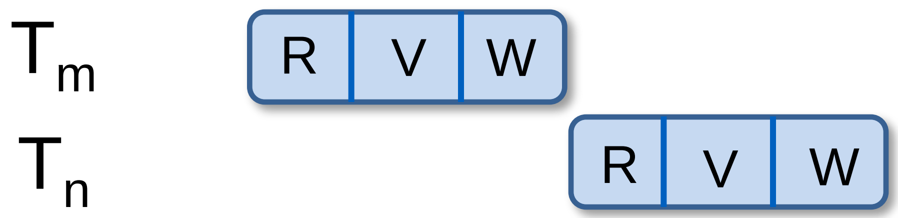
A: no ($r_m > w_n$) $\Leftarrow$ read_set(T_m) $\cap$ write_set(T_n) = $\varnothing$

B: no ($w_m > r_n$) $\Leftarrow$ write_set(T_m) $\cap$ read_set(T_n) = $\varnothing$

C: no ($w_m > w_n$) $\Leftarrow$ write_set(T_m) $\cap$ write_set(T_n) = $\varnothing$

Exercise 7.4
Optimistic Concurrency Control

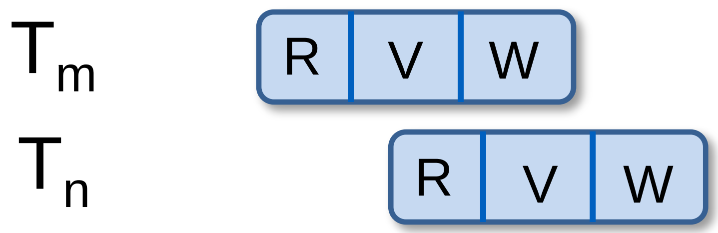
1. end_write_phase(T_m) < start_read_phase(T_n)



A, B, and C are always fulfilled, since op(T_m) < op(T_n)

Exercise 7.4
Optimistic Concurrency Control

2. end_write_phase(T_m) < start_write_phase(T_n)



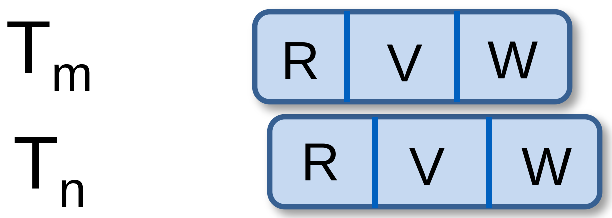
A: o.k., since x : $r_m(x) < w_n(x)$

B: must be checked, since $w_m[x] > r_n[x]$ is possible

C: o.k., since x : $w_m(x) < w_n(x)$

Exercise 7.4
Optimistic Concurrency Control

3. end_read_phase(T_m) < end_read_phase(T_n)



A: o.k., since $r_m[x] < w_n[x]$
B: must be checked, since $w_m[x] > r_n[x]$ is possible
C: must be checked, since $w_m[x] > w_n[x]$ is possible

Exercise 7.4
Optimistic Concurrency Control

TECHNISCHE UNIVERSITÄT DARMSTADT

Analysis of Case 1 - Case 3

Condition A doesn't need to be checked.

Case 1 was trivial, so we need to validate only against transactions T_m such that: **start_read_phase(T_n) < end_write_phase(T_m)**

If the condition of case 2 is fulfilled (i.e. no concurrent write phases are possible), it is enough to check that **write_set(T_m) ∩ read_set(T_n) = ∅.**

Exercise 7.4
Optimistic Concurrency Control

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Case 3 means concurrent write phases are possible**. In this case we must check:
  $write\_set(T_m) \cap read\_set(T_n) = \varnothing$
  $write\_set(T_m) \cap write\_set(T_n) = \varnothing$

  If *B* is fulfilled, but *C* is not fulfilled, then the write phase of T_n should be deferred to the end of T_m!