12 Concurrency control

12.1 Serializability and Concurrency Control

12.2 Locking

Lock protocols

- Two phase locking
- Strict transactional protocols
- Lock conflicts and Deadlocks
  - Lock modes
  - Deadlock detection, resolution, avoidance

12.3 Nonlocking concurrency control

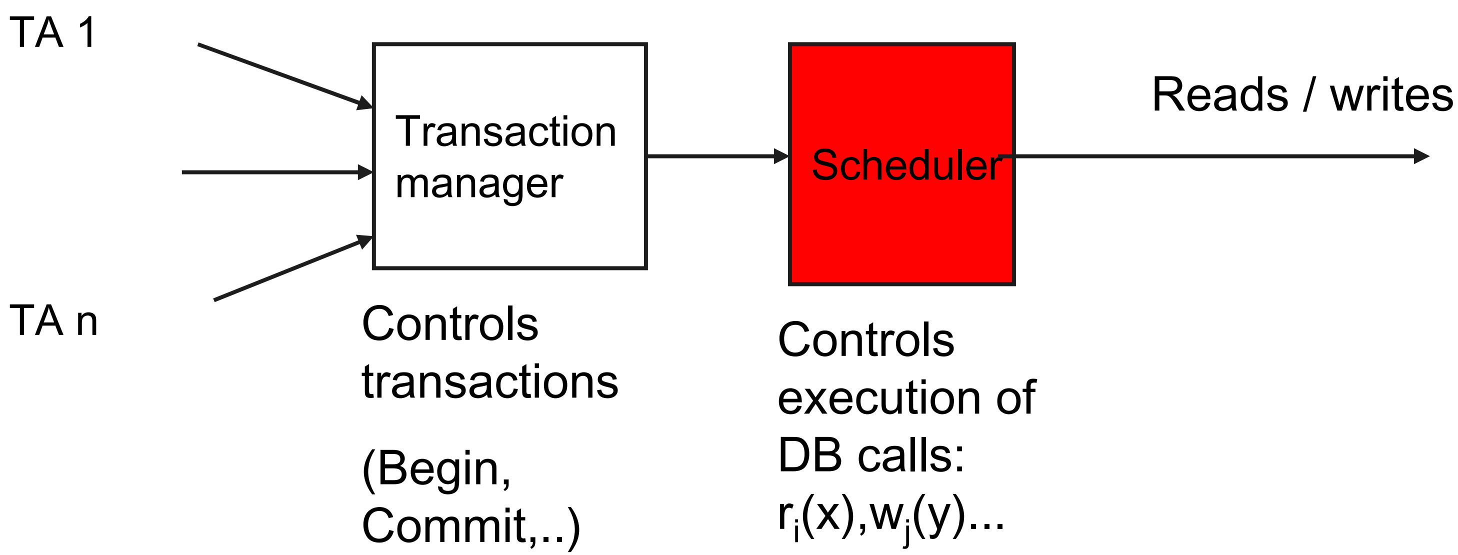
- 12.3.1 Time stamp ordering
- 12.3.2 Optimistic cc methods
  - 12.3.3 Multiversion cc

Lit.: Eickler/ Kemper chap 11.6-11.12, Elmasri /Navathe chap. 20, Garcia-Molina, Ullman, Widom: chap. 18

Concurrency control…and serializability

<u>Wanted:</u>

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



Concurrency control

Def.: Concurrency control (*) in DBS

Methods for **scheduling the steps (operations) of database transactions** in a way which isolates concurrent transactions in order to guarantee serializability.
("between system start and shutdown”).

(*) "Synchronisierung"

Methods

Approaches:

(1) **Pessimistic**: Scheduler has to check if next incoming operation can be executed without compromising isolation.

(2) **Optimistic:** Check for potential conflicts at the end of a TA. If yes: abort, else write effects into DB

(3) **Multiversion cc (MVCC)**: orthogonal to (1), (2).
More than one version of each data object allowed.
May employ (1) or (2).
Main advantage: readers preferred.
⇒ very useful for "read-mostly" databases

CC methods

Primary concurrency control **methods**

1. Locking (most important)

2. Non-locking protocols:

Optimistic concurrency control

Time stamps

Multiversion CC (locking and non-locking variants)

3. **MVCC** more and more important

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!

There are **read lock (shared locks)** and **write locks (exclusive)** - makes sense: **no read-read conflicts**

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

Optimistic vs. pessimistic

Locking is <u>pessimistic</u>

Scenario: during operation op (x) of TA1 a (potentially) conflicting operation op'(x) of TA2 may access the same object x .

Avoided by locking x before accessing this object.

T1: r1(u), w1(y)................... w1(y), w1(x) c1

T2: r2(y) w2(x) w2(z), c2

T3: r3(u) w3(x) c3

t

How long should a data objects be locked?

T12.2. Lock protocols

Freie Universität Berlin

Def.: **Standard locking**

1. Each object referenced by TA_i has to be locked before usage
2. Existing locks of other TA's will be respected i.e. wait until lock released.
3. Locks are released eventually.
4. A requests of a lock by a TA which it already holds, has no effect.
5. No preemption of a lock by another TA.

Lock protocols

Standard object locking does not help....

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

⇨ schedule may not be serializable (why?)

Example

l1(x) r1(x) ul1(x) l1(x) w1(x) ul(x)

l2(x) w2(x) ul2(x)

⇨ lost update ⇨ useless

l(x), ul(x)= lock / unlock x

Lock protocols

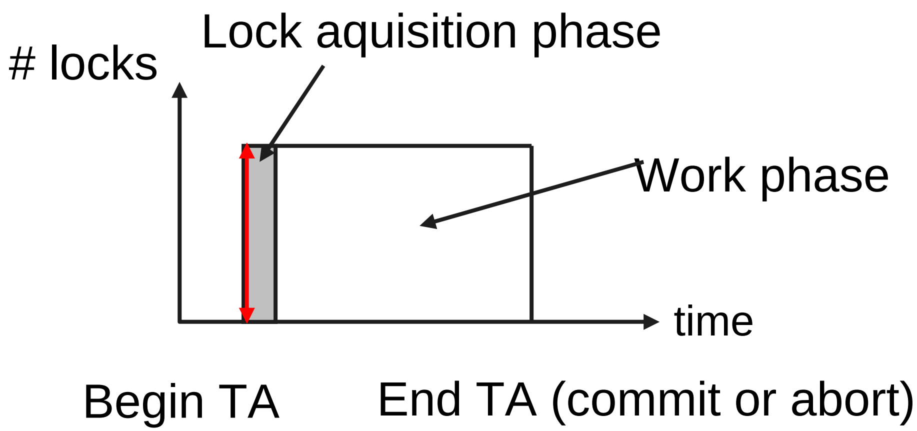
Freie Universität Berlin

Preclaiming

- Acquire <u>all locks</u> 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.

Two phase locking (2PL)

Freie Universität Berlin

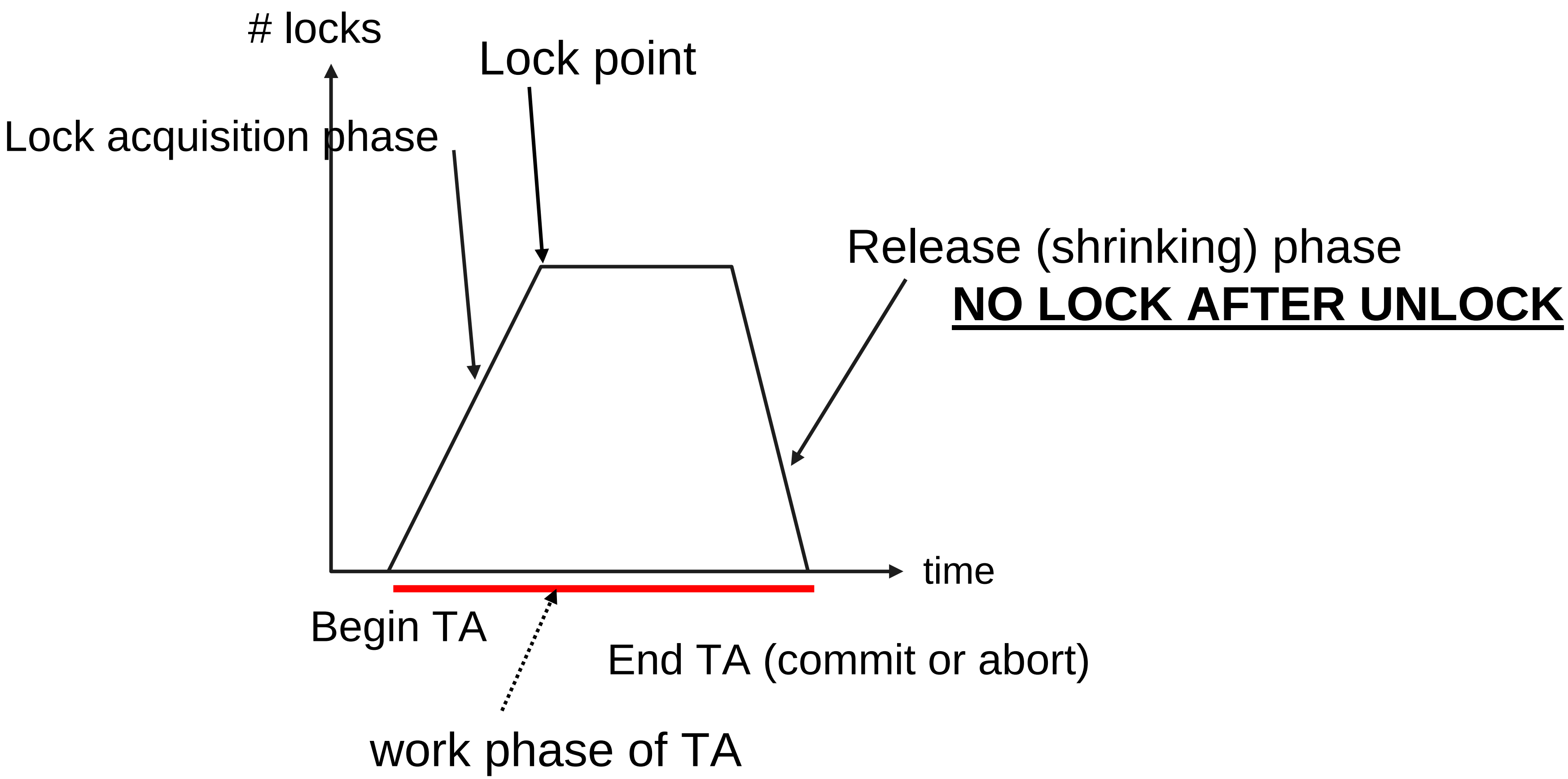
Def.: **Two phase locking**

1. All rules of standard locking hold.
2. If a transaction TA **releases a lock** it holds on an object, TA **must not aquire a new lock** on any object.

NO LOCK AFTER UNLOCK !

Standard method for concurrency control in many database systems

Concurrency control: 2PL



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

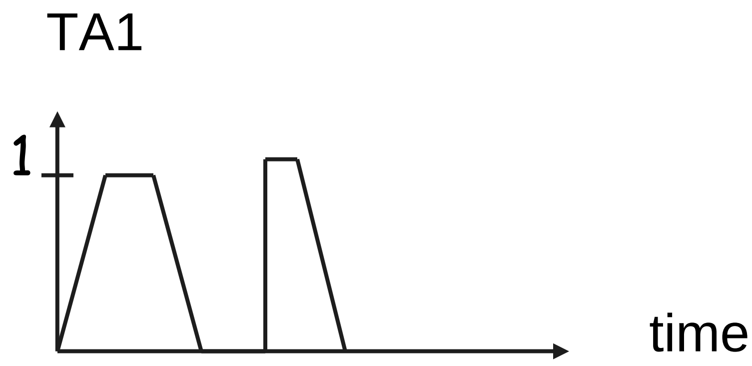
Concurrency control 2PL

Freie Universität Berlin

Why no lock after unlock?

	TA1	TA2
↓ t	l1(x)	
	r1(x)	
	x=x*10,	
	ul1(x)	
		l2(x),
		r2(x),
		x:=x+1,
		w2(x)
		ul2(x)
	l1(y)	
	w1(y=x+y)	
	ul1(y)	

inconsistent (wrong!) y



Lock profil of TA1 is NOT 2PL

2-Phase locking theorem

If all transactions follow the **2-phase locking** protocol, the resulting history is **serializable.**

Proof sketch:

Suppose a resulting schedule is not serializable.
$\Rightarrow$ conflict graph contains a cycle $\Rightarrow$ there are transactions TA1 and TA2 with conflict pairs (p,q) and (q', p').

$\Rightarrow$ One of the them must have violated the "no lock after unlock" rule

(assuming a cycle of length 1, induction for the general case)

→

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: l (y), T1: l (x), T2: l (x), T1: l (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 TAs use 2PL protocol

Same holds for the other possible sequences $\Rightarrow$ Theorem

2PL ≡ Serializable?

Freie Universität Berlin

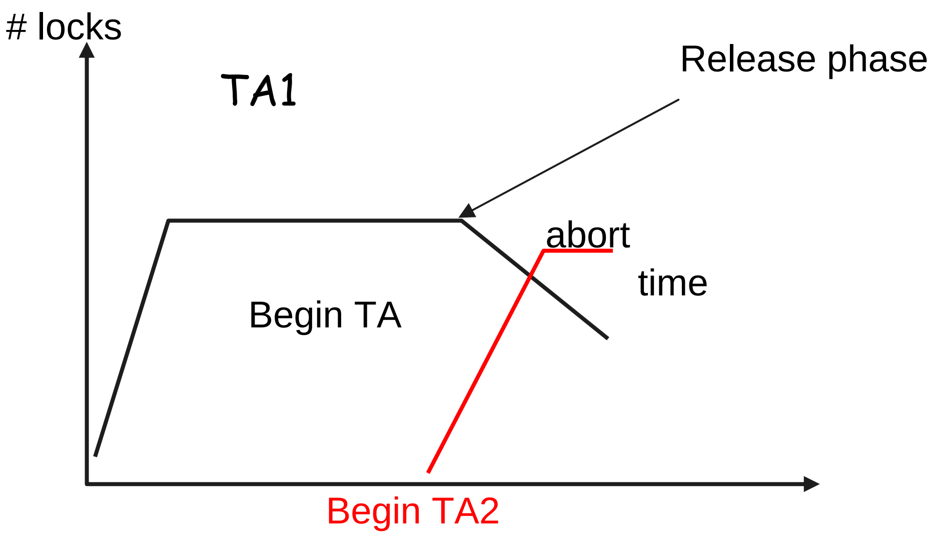
The **converse** of the 2PL theorem **does not hold**:

There are histories which are serializable, but the TAs did not lock according to 2 PL.

l1(x)r1[x] ul1(x) l2(x) w2[x] ul2(x) c2 l(y) w1[y] ul(y) c1

Conflict graph

Strict concurrency protocols: motivation

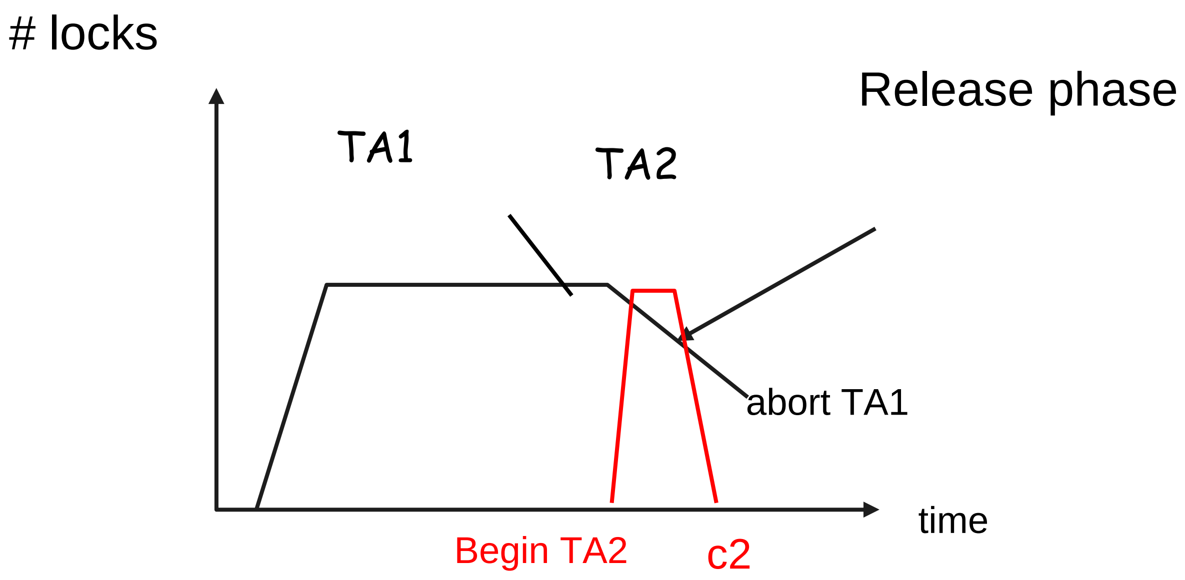


A different transaction TA2 could have used an object x which was unlocked by TA1 in the release phase.
... no problem, if TA1 commits,

but abort... ??

Recursive situation $\Rightarrow$ **cascading abort**

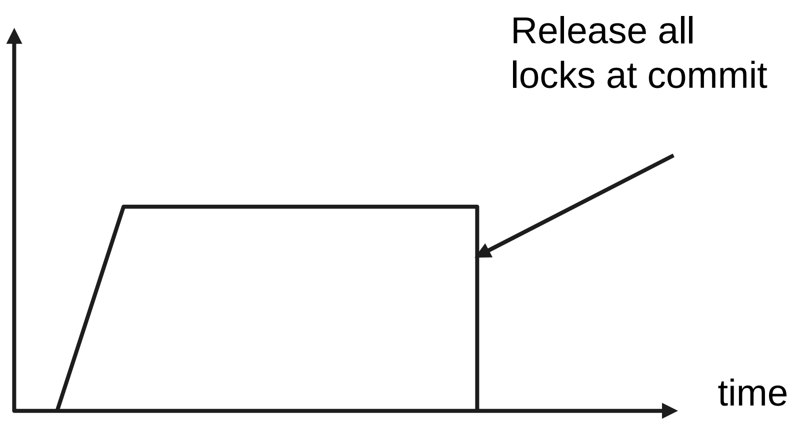
Nonstrict example: even worse



Worse situation, why?

Strict 2PL

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



Strict 2PL protocol

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: e.g. **time stamps**, optimistic cc

Lock modes

Freie Universität Berlin

Lock modes and lock compatibility

RW-(SX) – model: read (R) and eXclusive(W) locks
(or: write locks)

holder

requester	R	W
R	+	-
W	-	-

R-lock =
Shared (S) lock
W-lock = X-lock

Lock compatibility matrix

Lock compatibility in the RW model:

Objects locked in R-mode may be locked in R- mode by other transactions(+)

Objects locked in W-mode may not be locked by any other transaction in any mode.
Lock conflict ⇒ requesting TA has to wait

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

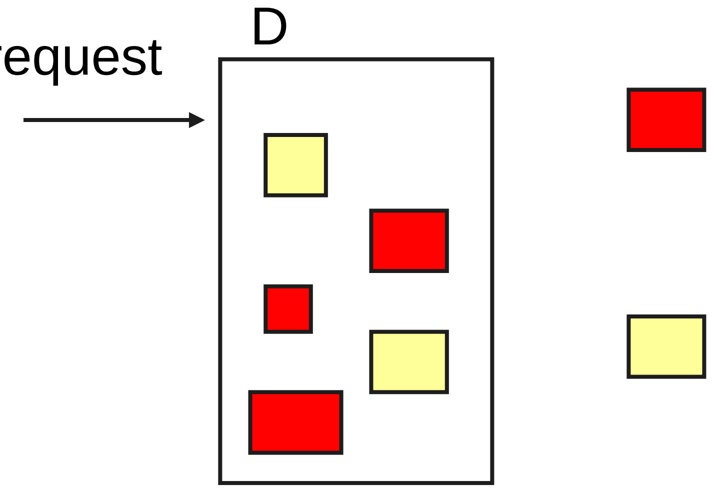
- some rows of R locked by different transactions

⇨ 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

Locks of different granularity

Lock request for D



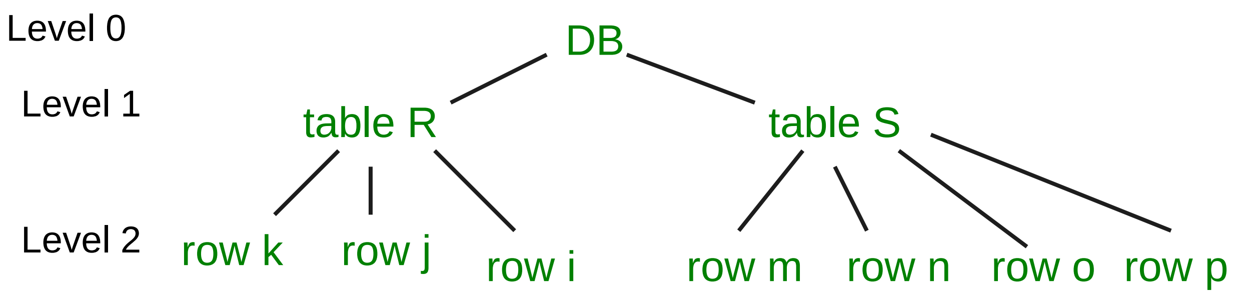
Lock held by other transactions$_n$

Lock request: must not be granted until lock on D is released

Efficient implementation of this type of situation??

Lock modes: Hierarchical locking

Intention locks



Object hierachy (example)

Page 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 an Intention_ρ-lock on an object D on level i, if and only if it holds an ρ-lock on an object D' on level j > i subordinate to D

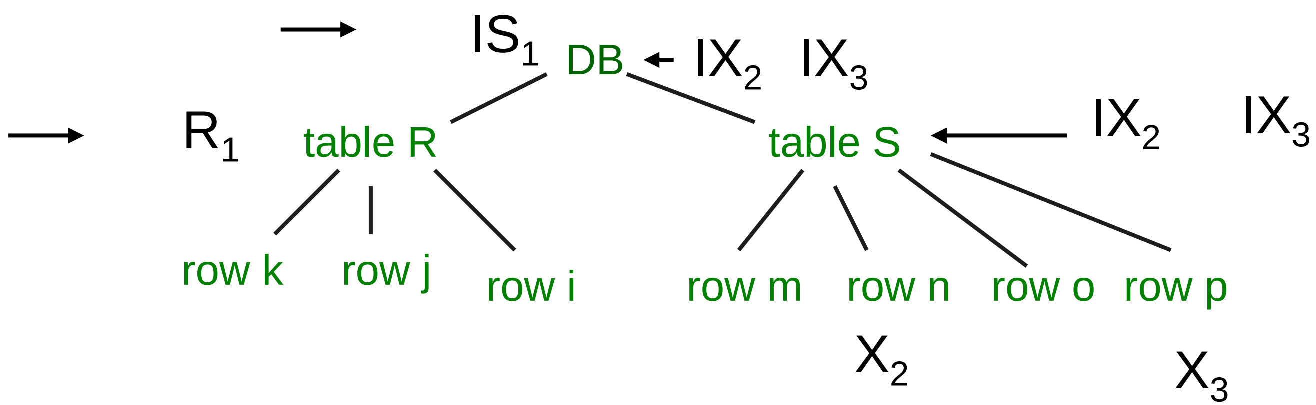
Concurrency control

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

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: simply implement I<M>-lock as a counter

holder (columns) / requester (rows)

	IR	IX	R	X
IR	+	+	+	-
IX	+	+	-	-
R	+	-	+	-
X	-	-	-	-

Compatibility matrix

Lock conflicts and deadlocks

Freie Universität Berlin

Lock conflict

Two or more processes request an exclusive lock for the same object

Deadlock (*)

Locking: always threat of deadlock if

- 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:

$l_1(x)$, $l_2(y)$, ..., $l_1(y)$, $l_2(x)$ causes deadlock

(*) Verklemmung

Deadlock detection and resolution

Freie Universität Berlin

Deadlocks

Release of a lock could break rule 4

wl1(x) , wl2(y), wl1(y) -> TA1: WAIT for wu2(y) , wl2(x) -> TA2: WAIT for wu1(x)

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

... wl1(x) , wl2(y), wl1(y) -> TA1: WAIT for wu2(y) wl2(z), w2(y), w2(z), wu2(y),...

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

Lock conflict resolved by wu2(x), TA1 proceeds

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

Deadlock

Def.: Wait-for graph WF = (T, E) where
- node set T is the set of running transactions
- edges $(T_i, T_j) \in E \Leftrightarrow T_i$ required a lock on some object x which has been locked by T_j in an incompatible mode.

There is a deadlock between transactions
$\Leftrightarrow$ WF contains a cycle

$\Rightarrow$ System has to check WF for cycles periodically.

Deadlock resolution

Freie Universität Berlin

Resolving deadlocks

In case of cycle:

- One of the waiting transaction ("victim") has to be rolled back
- Which one? Heuristic decision: youngest, TA with least write activity, ...

Timeout: an alternative?

- If TA has been waiting longer than the time limit, it is aborted.
- No: efficient but **may roll back innocent victims** (deadlock does not exist)

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