SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (4/6)

- query optimization
  - query rewrite
  - cost-based
- data recovery
  - quick recap of transaction management
  - single instance recovery: ARIES
- transaction handling
  - scheduling of transaction operations
  - concurrency control
  - implementing isolation levels
- parallelization of data and queries
  - horizontal partitioning, vertical partitioning, replication
  - distributed query processing
  - multi-cores
  - map-reduce

Transaction Handling.

Motivation: Concurrency

- the DBMS may decide to interleave atomic operations of different transactions
- this usually happens when the DBMS allows for concurrent execution of transactions
- concurrency is used for two reasons:
  - hide latency
    - for example: if one TA waits for the disk, others my do useful work
    - long TAs should not block short TAs
    - for example: long analytic query should not block short primary access path queries
- in the following we will think about how to interleave operations of different transactions
- goal: interleaving these operations should not lead to “unexpected“ results

Motivation: Batched-Processing

- interleaving of TAs may also make sense in the **single-threaded case!**
- For example:
  - we may decide to collect a set of transactions before executing
  - then we execute transactions in a batch however interleaving execution of different transaction operations cleverly
  - goal: make use of data and cache locality
  - price: latency (the bigger the better in this case)
  - same phenomenon as batch processing on index structures
  - no multi-threading whatsoever here
  - **may be useful for project**
- **Note: the same optimization techniques apply for both concurrency and batched processing**

Lost Updates/Race Conditions

step	T_1	T_2
1.	$\text{read}(A, a_1)$	
2.	$a_1 := a_1 - 300$	
3.		$\text{read}(A, a_2)$
4.		$a_2 := a_2 * 1.03$
5.		$\text{write}(A, a_2)$
6.	$\text{write}(A, a_1)$	
7.	$\text{read}(B, b_1)$	
8.	$b_1 := b_1 + 300$	
9.	$\text{write}(B, b_1)$	

- changes of T_2 are lost

Dependency on Non-committed Writes

step	T_1	T_2
1.	$\text{read}(A, a_1)$	
2.	$a_1 := a_1 - 300$	
3.	$\text{write}(A, a_1)$	
4.		$\text{read}(A, a_2)$
5.		$a_2 := a_2 * 1.03$
6.		$\text{write}(A, a_2)$
7.	$\text{read}(B, b_1)$	
8.	$\ldots$	
9.	**abort**	

- transaction T_2 reads data that was never comitted (data that should never have existed)
- this is also called “dirty read”

Phantom Problem

T_1	T_2
	select sum(balance)
	from accounts
insert into accounts	
values $(C, 1000, \ldots)$	
	select sum(balance)
	from accounts

- T_1 inserts a new row into table accounts while T_2 is running
- T_2 executes the same SQL statement twice with different results
- the newly inserted row is called “phantom“
- this problem happens when data is selected based on a predicate

Serial Execution of T_1 and T_2: T_1 | T_2

step	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.	read(B)	
5.	write(B)	
6.	**commit**	
7.		**BOT**
8.		read(C)
9.		write(C)
10.		read(A)
11.		write(A)
12.		**commit**

Serializability

- a schedule is called serializable if the schedule is equivalent to a serial schedule $T_1 | T_2$ or $T_2 | T_1$
- parallel execution possible
- database is in the same state as if the serial schedule were executed
- Note:
  - serializability does not specify any particular serial order
  - we are only saying that the execution is equivalent to **some** serial order
  - order of atomic operations inside a TA may never be changed

step	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.		**BOT**
4.		read(C)
5.	write(A)	
6.		write(C)
7.	read(B)	
8.	write(B)	
9.	**commit**	
10.		read(A)
11.		write(A)
12.		**commit**

serializable schedule of T1 and T2

Non-Serializable Schedule

- T_1 accesses data item A before T_3
- T_3 accesses data item B before T_1
- this schedule is not equivalent to any of the serial schedules $T_1 | T_3$ or $T_3 | T_1$

step	T_1	T_3
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.		**BOT**
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		**commit**
10.	read(B)	
11.	write(B)	
12.	**commit**	

Two Parallel Transactions (1/2)

- Note: this schedule is **not** serializable
- However, in this special case the database would be consistent!
- This is just by chance!
- DBMS does not know about the semantics of the application.
- DBMS only “sees“ read and write operations

step	T_1	T_3
1.	**BOT**	
2.	$\text{read}(A, a_1)$	
3.	$a_1 := a_1 - 50$	
4.	$\text{write}(A, a_1)$	
5.		**BOT**
6.		$\text{read}(A, a_2)$
7.		$a_2 := a_2 - 100$
8.		$\text{write}(A, a_2)$
9.		$\text{read}(B, b_2)$
10.		$b_2 := b_2 + 100$
11.		$\text{write}(B, b_2)$
12.		**commit**
13.	$\text{read}(B, b_1)$	
14.	$b_1 := b_1 + 50$	
15.	$\text{write}(B, b_1)$	
16.	**commit**	

Two Parallel Transactions (2/2)

- Note: this schedule is **not** serializable
- However, in this special case the database would **not** be consistent!
- Here, the semantics of the application makes the databases inconsistent.
- this schedule is neither equivalent to the serial schedules $T_1|T_3$ or $T_3|T_1$
- The DBMS does not know about the application semantics.

step	T_1	T_3
1.	**BOT**	
2.	$\text{read}(A, a_1)$	
3.	$a_1 := a_1 - 50$	
4.	$\text{write}(A, a_1)$	
5.		**BOT**
6.		$\text{read}(A, a_2)$
7.		$a_2 := a_2 * 1.03$
8.		$\text{write}(A, a_2)$
9.		$\text{read}(B, b_2)$
10.		$b_2 := b_2 * 1.03$
11.		$\text{write}(B, b_2)$
12.		**commit**
13.	$\text{read}(B, b_1)$	
14.	$b_1 := b_1 + 50$	
15.	$\text{write}(B, b_1)$	
16.	**commit**	

Recoverable Schedules

- Goal: every transaction may be rolled back locally
- a schedule is recoverable if in any case the writing transaction $T_{j \neq i}$ commits before the reading transaction T_i.
- in other words:
  a transaction T_i may only commit if all transactions $T_{j \neq i}$ that modified data that was read by T_i have committed.
- If this rule is violated, it may become impossible to locally rollback a transaction as T_i has used data values that “officially“ never existed in the database.

Cascading Aborts

- T_2 reads from T_1, T_3 reads from T_2, etc.
- in step 9 T_1 aborts
- Effect: all transactions T_2 - T_5 have to be aborted and rolled back! (snowball effect)

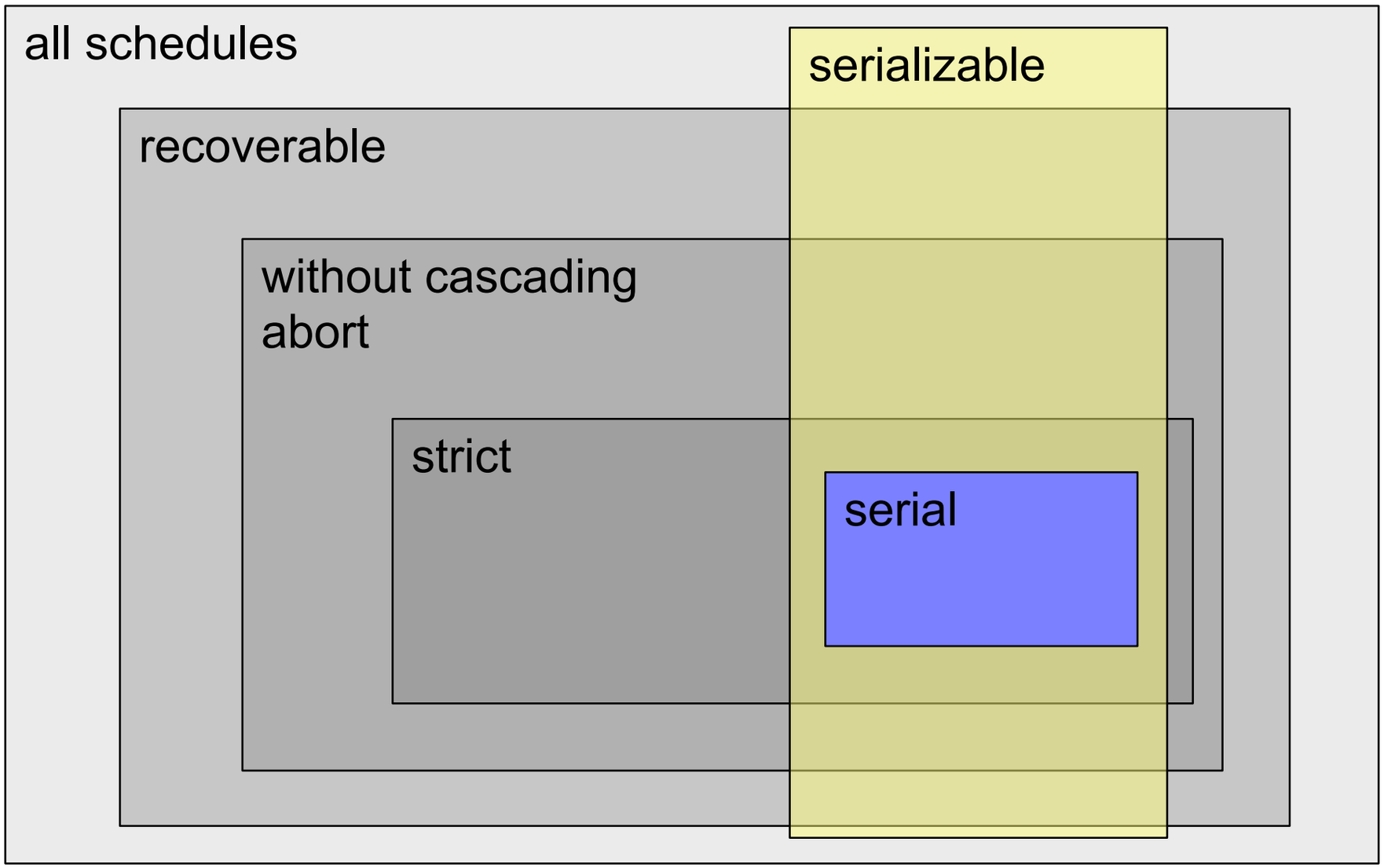
step	T_1	T_2	T_3	T_4	T_5
0.	$\cdots$				
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_4(D)$	
8.					$r_5(D)$
9.	a_1 **(abort)**				

- How to obtain a schedule without cascading abort:
  - schedule is a recoverable schedule
  - read changes from other transactions only after these transactions have committed.

Strict Schedules

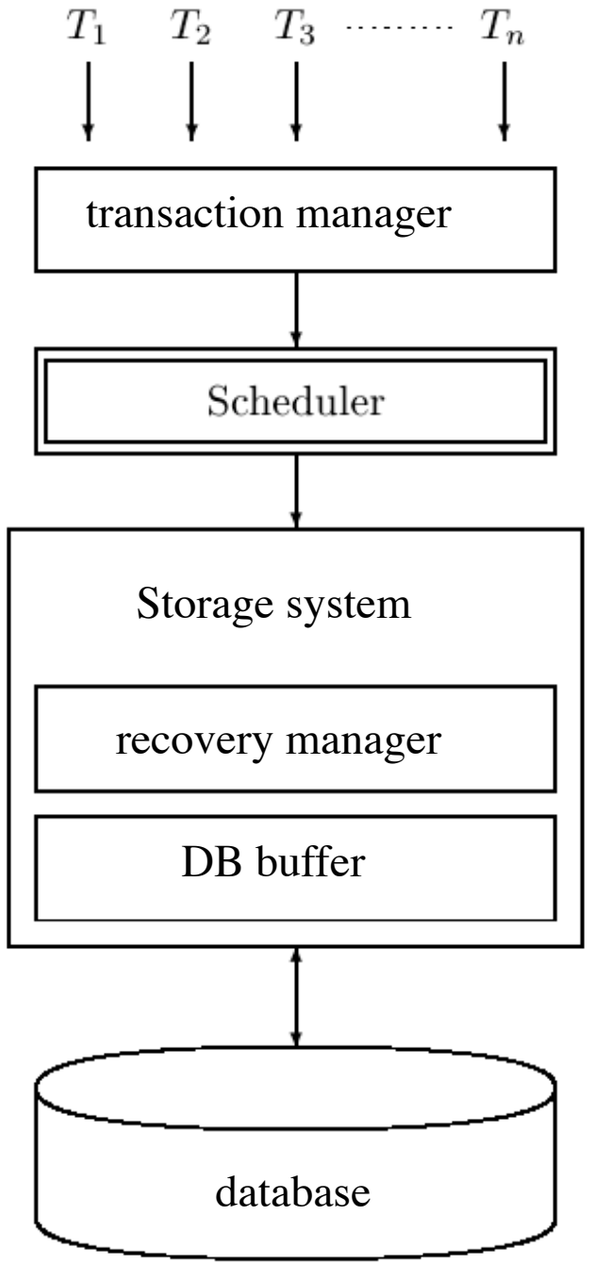
- A strict schedule is a schedule
  - without cascading rollbacks
  - and: data that was changed by a transaction that did not commit or abort yet, may not be overwritten
- For instance:
  is a transaction T_1 changes a value A, no other transaction running in the system may read or overwrite this value until T_1 committed.
- we will come back to this
  (avoided by using strict two-phase locking)

Relationship Among Different Schedules

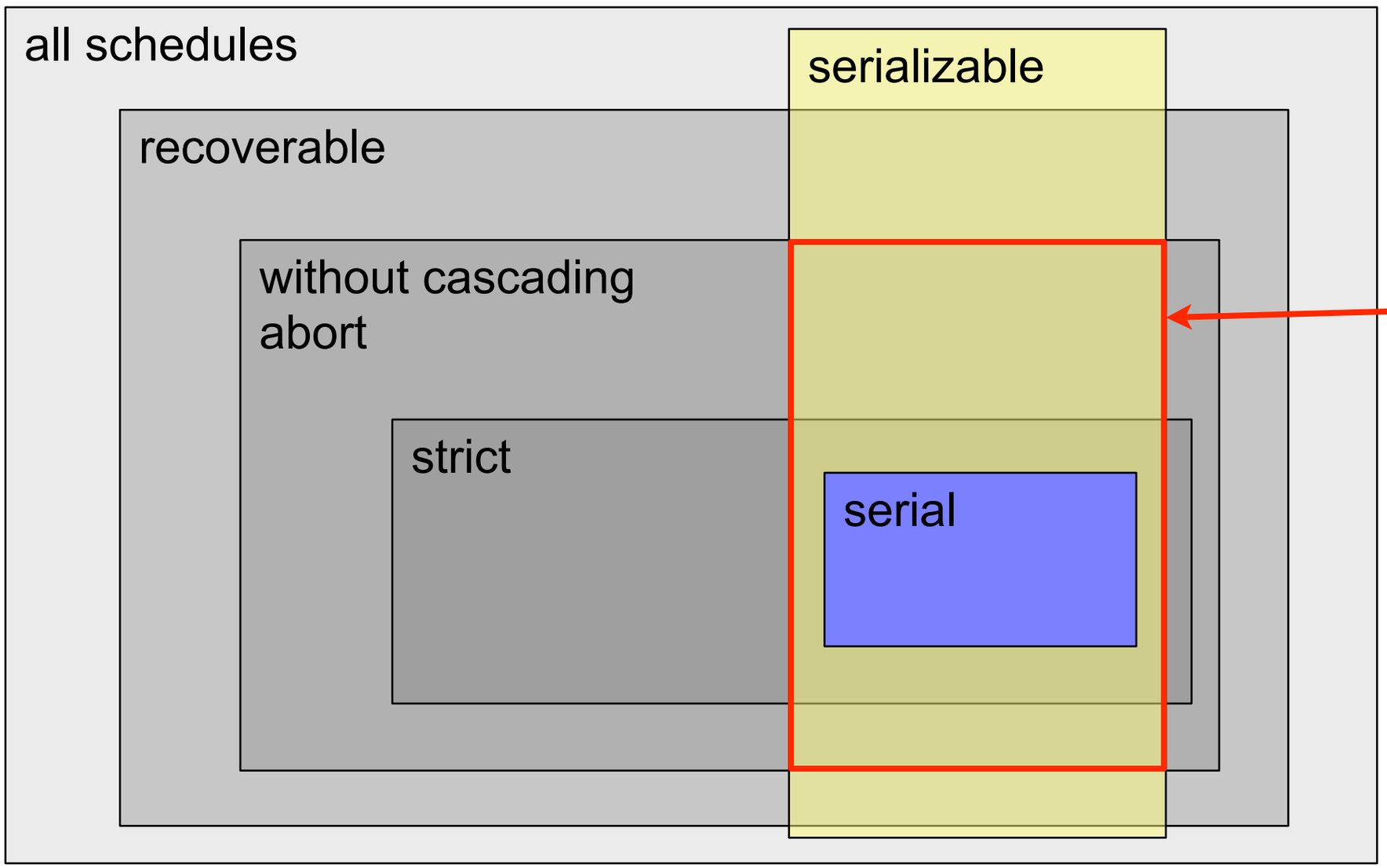


Database Scheduler

- has to schedule operations in a way such that the resulting schedule is serializable
- in addition: schedules should also be without cascading aborts, i.e,
- in the following: techniques for realizing a database scheduler



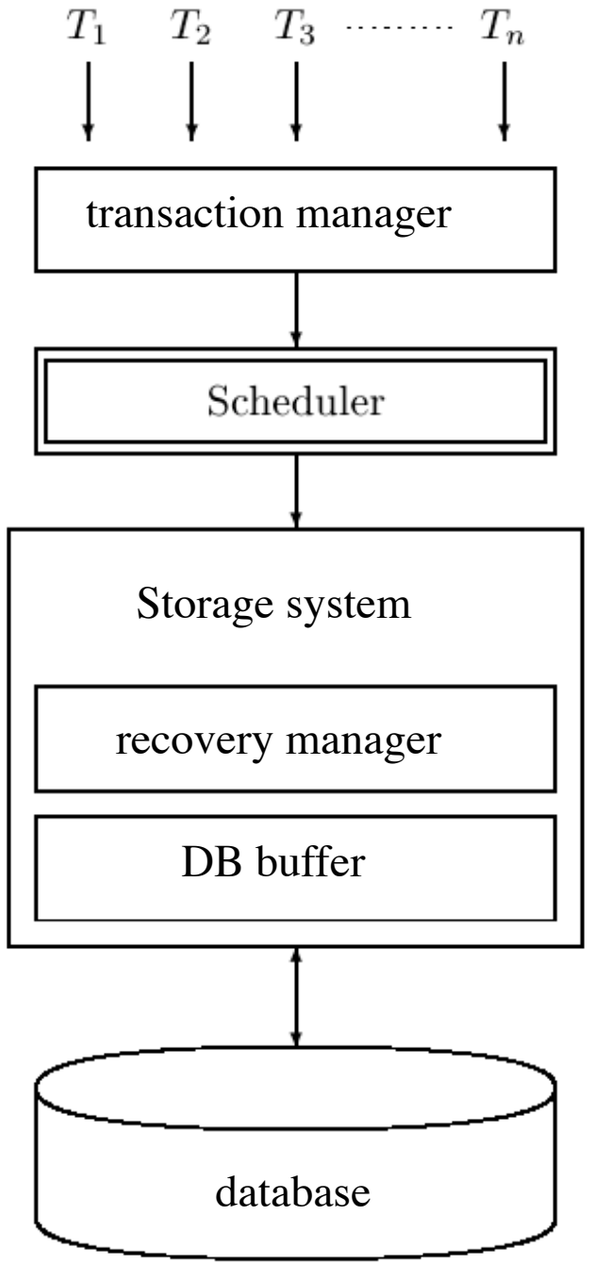
Relationship Among Different Schedules



scheduler has to compute these schedules, i.e., intersection of serializable schedules and schedules without cascading abort

Implementing the Database Scheduler

- pessimistic approaches ("it will go wrong in most of the cases")
  - lock-based
  - timestamp
- optimistic approach ("it will work in most of the cases")
  - validate after commit



Lock-based Synchronization

- two types of locks
  - S (**S**hared lock, read)
  - X (e**X**clusive lock, write)
- compatibility matrix

existing lock

requested lock

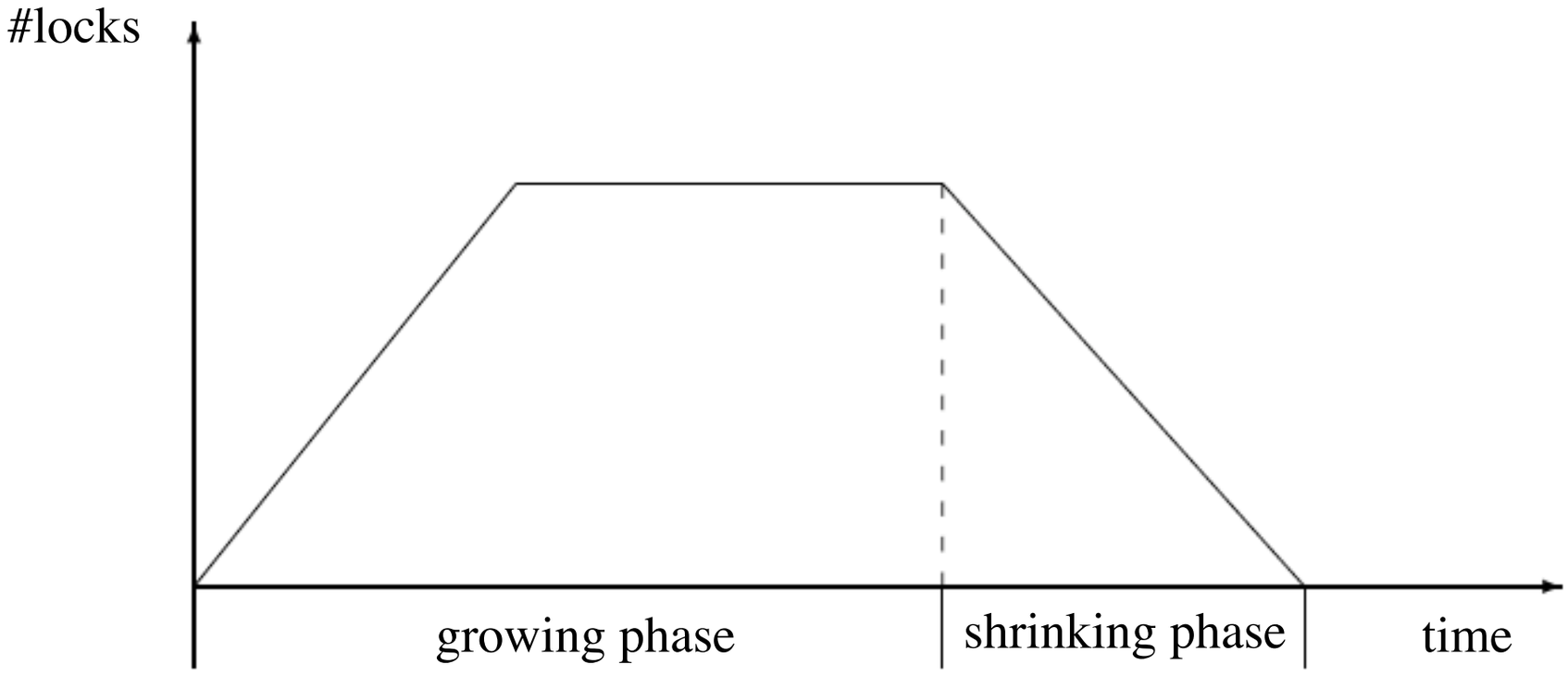
	NL	S	X
S	$\surd$	$\surd$	–
X	$\surd$	–	–

NL: no existing lock

Two Phase Locking (2PL)

- every object used by a TA has to be locked before being used
- a TA never requests a lock for an object that it has already locked
- the TA respects the locking scheme based on the compatibility matrix
- if a certain lock may not be obtained, the TA is suspended until the lock becomes available
- when the TA commits/aborts all its locks have to be released
- a TA operates in two phases
  - a **growing phase**: in this phase locks may be requested but may not be released
  - a **shrinking phase**: in this phase locks may not be requested, locks may only be released
  - at EOT all locks have to be released

2PL



- 2PL **guarantees** serializability
- 2PL is sufficient but not necessary for implementing serializability
- there are schedules that are serializable but would not be allowed by 2PL
- example: $r_0[A]->w_0[A]->r_1[A]->r_0[B]->w_0[B]->c_0->r_1[B]->c_1$
- which is equivalent to $T_0 | T_1$

Parallel Execution of TAs Respecting 2PL

- T_1 modifies data items A and B (e.g., a money transfer)
- T_2 reads data items A and B (e.g., compute sum of account balances)

step	T_1	T_2	note
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockS**(A)	T_2 has to wait
7.	**lockX**(B)		
8.	read(B)		
9.	**unlockX**(A)		T_2 continues
10.		read(A)	
11.		**lockS**(B)	T_2 has to wait
12.	write(B)		
13.	**unlockX**(B)		T_2 continues
14.		read(B)	
15.	**commit**		
16.		**unlockS**(A)	
17.		**unlockS**(B)	
18.		**commit**	

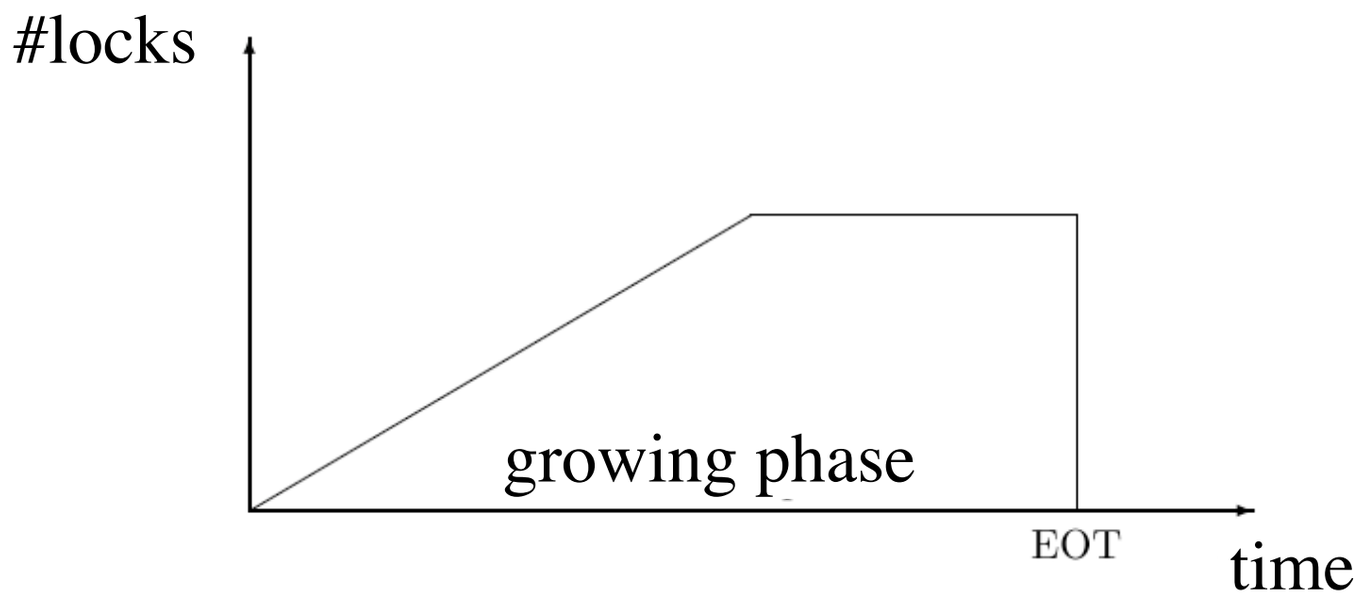
Cascading Abort?

- What happens if T_1 aborts before step 15?
- Then: also T_2 has to be rolled back as T_2 read 'dirty' data from T_1 in steps 10 and 14!
- Therefore 2PL does **not** avoid cascading aborts!

step	T_1	T_2	note
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockS**(A)	T_2 has to wait
7.	**lockX**(B)		
8.	read(B)		
9.	**unlockX**(A)		T_2 continues
10.		read(A)	
11.		**lockS**(B)	T_2 has to wait
12.	write(B)		
13.	**unlockX**(B)		T_2 continues
14.		read(B)	
15.	**commit**		
16.		**unlockS**(A)	
17.		**unlockS**(B)	
18.		**commit**	

Strict Two Phase Locking (Strict 2PL)

- Problem: 2PL does not avoid cascading aborts
- extension to **strict** 2PL
  - all locks will be held until EOT
  - no shrinking phase anymore: no locks will be released before EOT
  - this guarantees that cascading rollbacks may never happen
  - strict 2PL is the most widely used locking protocol
  - however: as with non-strict 2PL we have to handle **deadlocks**



Deadlocks

- Example schedule running into a deadlock (although respecting strict 2PL)

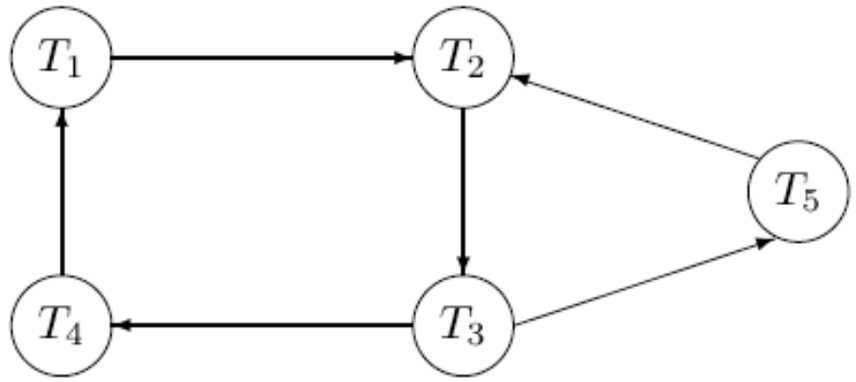
step	T_1	T_2	note
1.	**BOT**		
2.	**lockX**(A)		
3.		**BOT**	
4.		**lockS**(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	**lockX**(B)		T_1 has to wait for T_2
9.		**lockS**(A)	T_2 has to wait for T_1
10.	...	...	=> deadlock

Deadlock Detection: Timeout

- Timeout approach
  - if TA has not reacted for a certain time interval, abort TA
- Problem: how to chose time interval
  - too small:
    - too many TAs will be aborted
    - some of them may not have been in a deadlock but may have just been waiting for resources
  - too large:
    - deadlocks may go undetected for too long time
    - deadlocks degrade overall system performance

Deadlock Detection: Waits-for-Graph

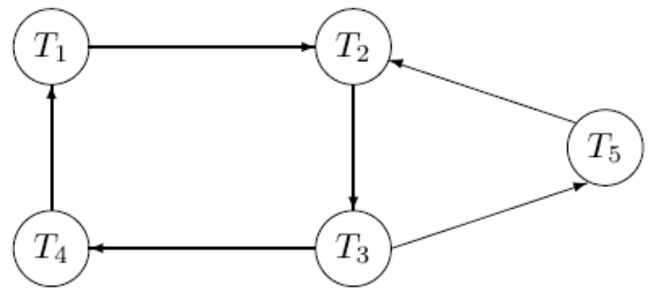
- idea: collect waiting TAs due to lock conflicts in a graph



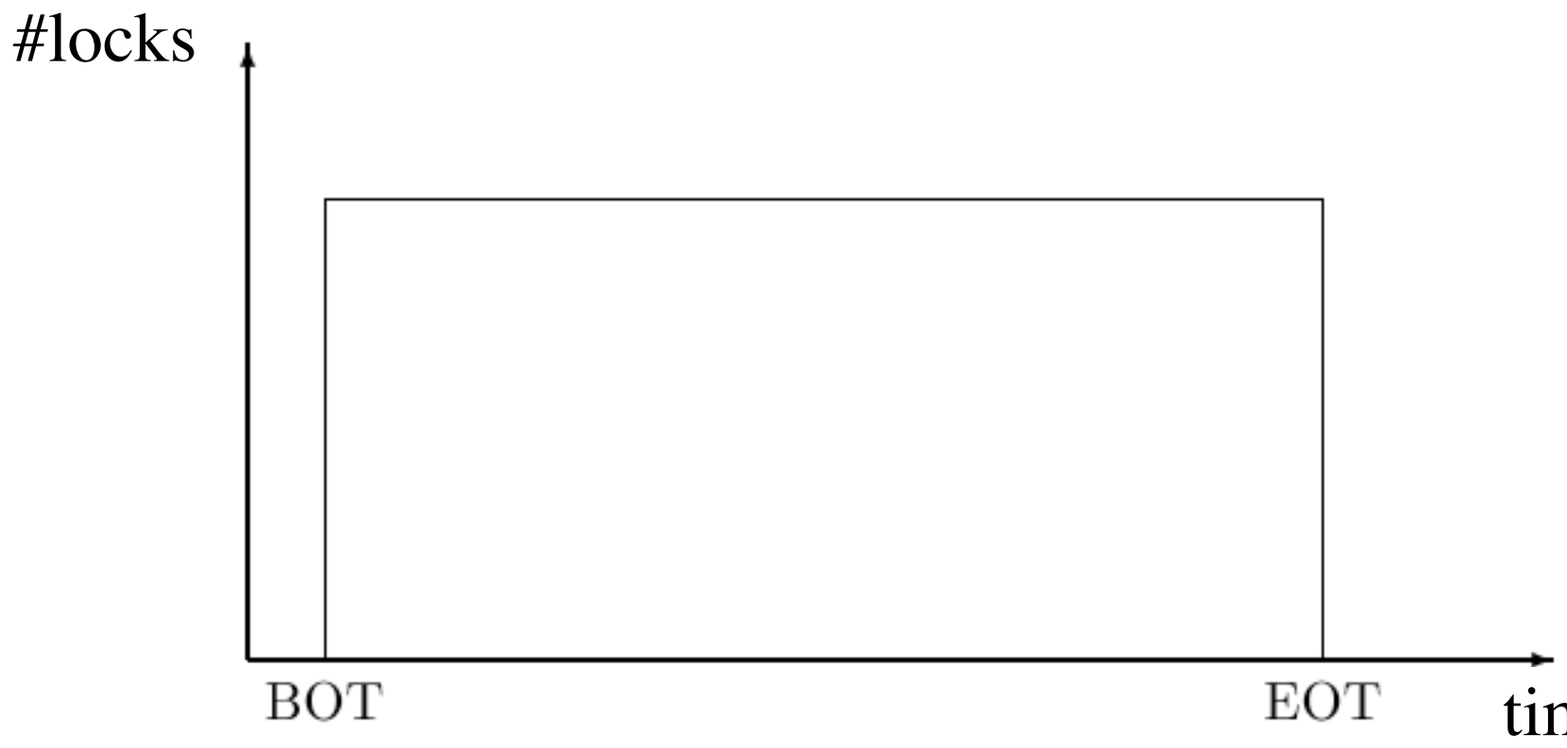
- if TA T_i waits for a lock held by T_j, we insert an edge T_i -> T_j
- waits-for-graph has two cycles
  - T_1 -> T_2 -> T_3 -> T_4 -> T_1
  - T_2 -> T_3 -> T_5 -> T_2
  - both cycles may be removed by aborting a single transaction T_3

Deadlock Detection: Which TA to abort?

- Which transaction of a detected cycle should be aborted?
- Either:
  - youngest TA or TA with smallest number of locks
  - maximize available resources: abort TA having the highest number of locks
  - avoid starvation: make sure that a TA that was aborted to resolve a deadlock before is not aborted again
    - avoid starvation of this TA
    - label TAs to track whether a TA was already aborted by the scheduler
  - multiple cylces: if a TA is part of multiple cylces, we may break multiple cycles at the same time by aborting a single TA (like in the example on the previous slide)



Deadlock Avoidance: Preclaiming



- how to avoid deadlocks in the first place?
- idea: preclaim all required locks before starting the TA
- problem how to know in advance which data items are required?
- therefore hard to realize in practice
- analogy: preplanning in the DB-buffer (see Week 1, Slide 128)

Deadlock Avoidance by Timestamping

- every TA receives a timestamp (TS)
- TAs may not wait indefinitely anymore to obtain a lock
- assume T_1 requests a lock that is held by T_2
- **wound-wait** strategy

  If $TS(T_1) < TS(T_2)$: //"T_1 older than T_2?"

  T_2 is **wounded**, T_2 is aborted and rolled back (however, keeping its timestamp)

  T_1 obtains lock

  Else //"T_1 younger than T_2?"

  T_1 **waits** until T_2 releases lock

prefer older TAs

- **wait-die** strategy

  If $TS(T_1) < TS(T_2)$: //"T_1 older than T_2?"

  T_1 **waits** until T_2 releases lock

  Else //"T_1 younger than T_2?"

  T_1 **dies**, i.e., T_1 is aborted and rolled back (however, keeping its timestamp)

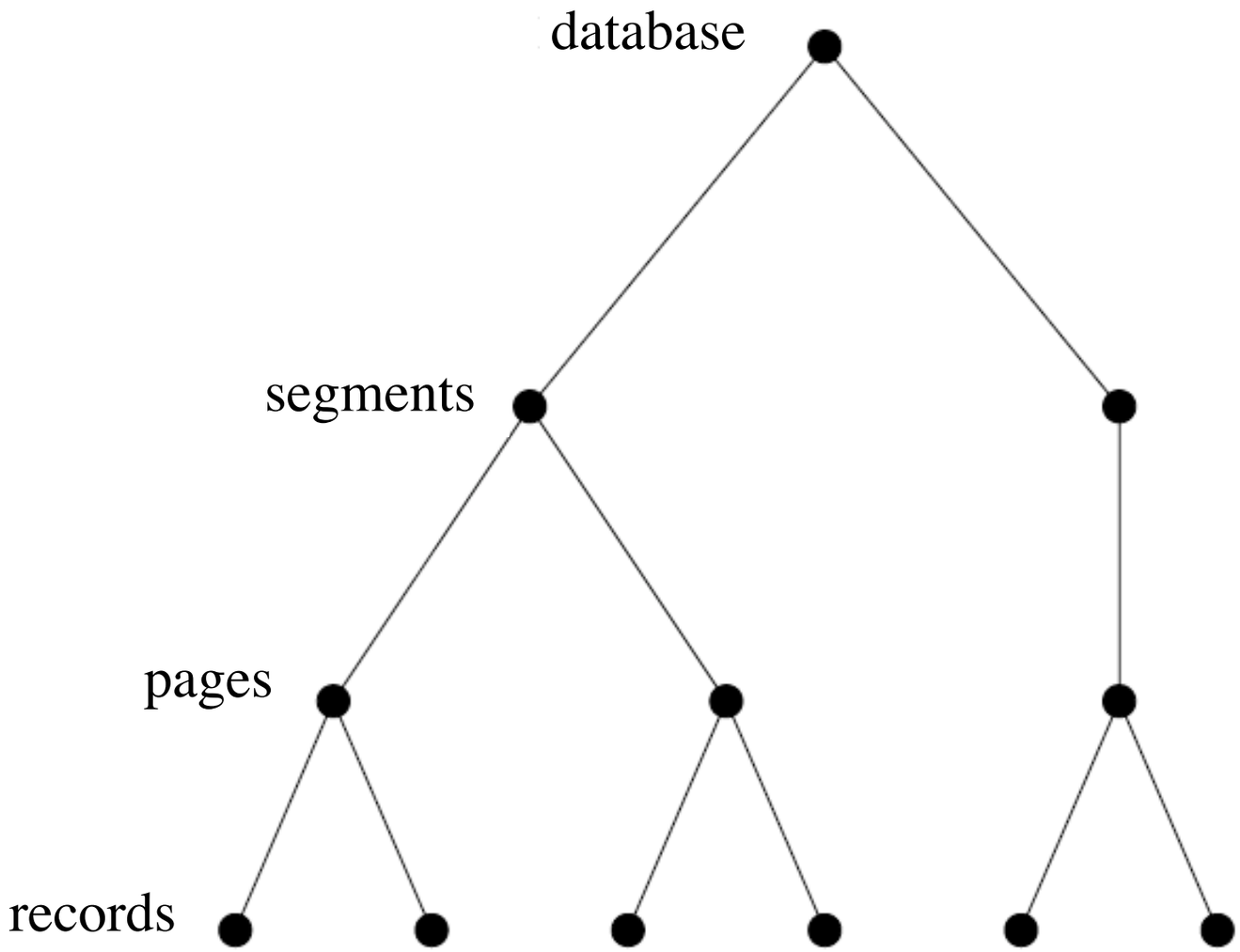
prefer younger TAs

- In both algorithms the younger TA is aborted!

Deadlock Avoidance by Timestamping

- method is guaranteed to resolve any deadlock
- wound-wait: prefer older TAs and never wait for younger TAs
- wait-die: prefer younger TAs and kill older TAs
- one of these methods or a combination may be selected to implement a deadlock prevention algorithm

Multi-Granularity Locking



- locks of different size, hierarchical overview
- analogy: reserving seats in the student‘s canteen

Extended Lock-types MGL

- NL: lock
- S (**S**hared lock, read)
- X (e**X**clusive lock, write)
- **IS (intention share)**: further down in the hierarchy a TA intends to obtain a shared lock
- **IX (intention exclusive)**: further down in the hierarchy a TA intends to obtain an exclusive lock

MGL: Lock Protocol

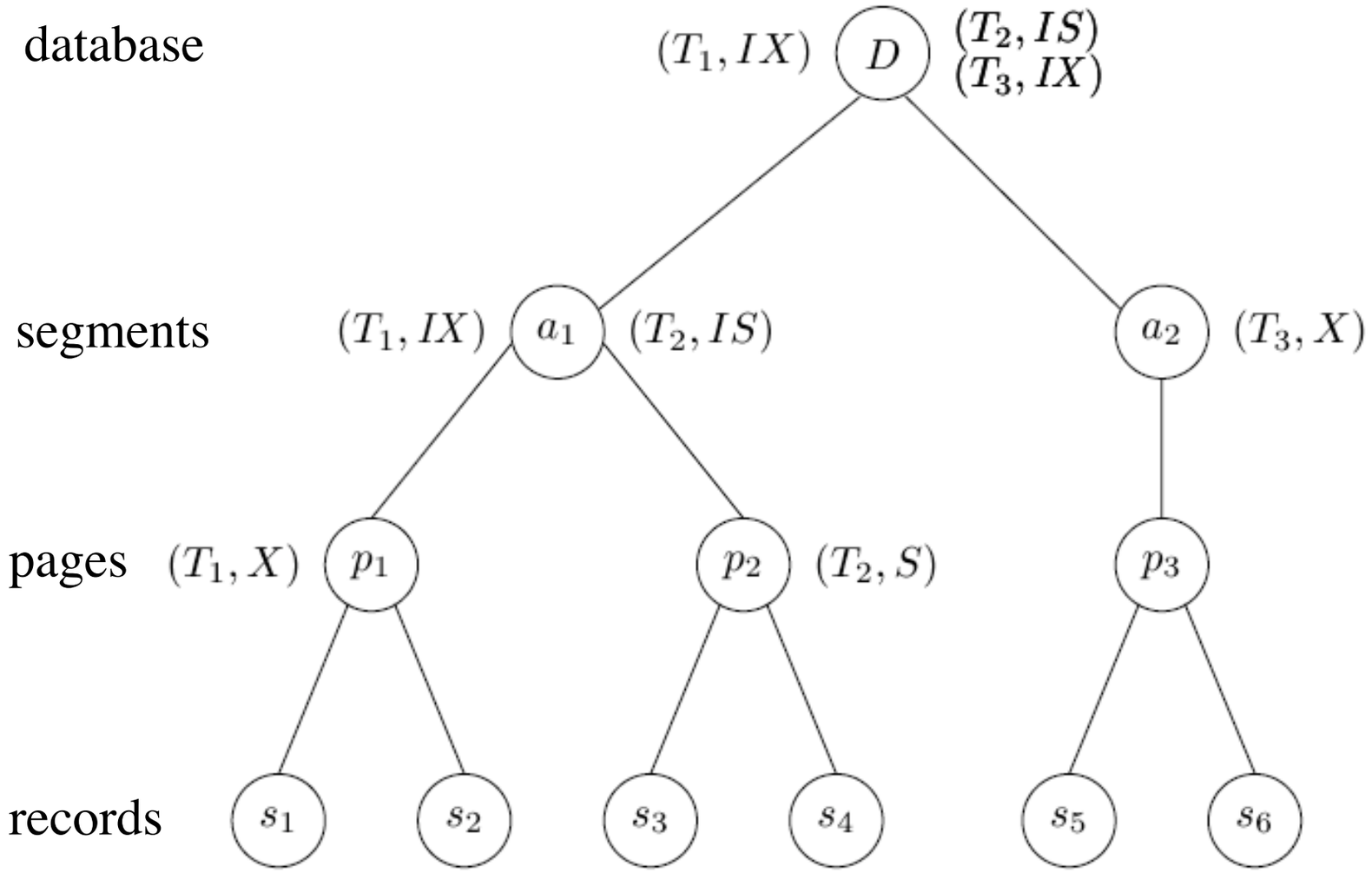
- locks are obtained top-down
  - before a node may be locked by S or IS, all predecessors in the hierarchy have to be locked by IX- or IS-locks
  - before a node may be locked by X or IX, all predecessors in the hierarchy have to be locked by IX-locks
- locks are released bottum-up:
  - before a lock on a node is released, all corresponding locks on successor nodes are released
- compatability matrix

existing lock

requested lock

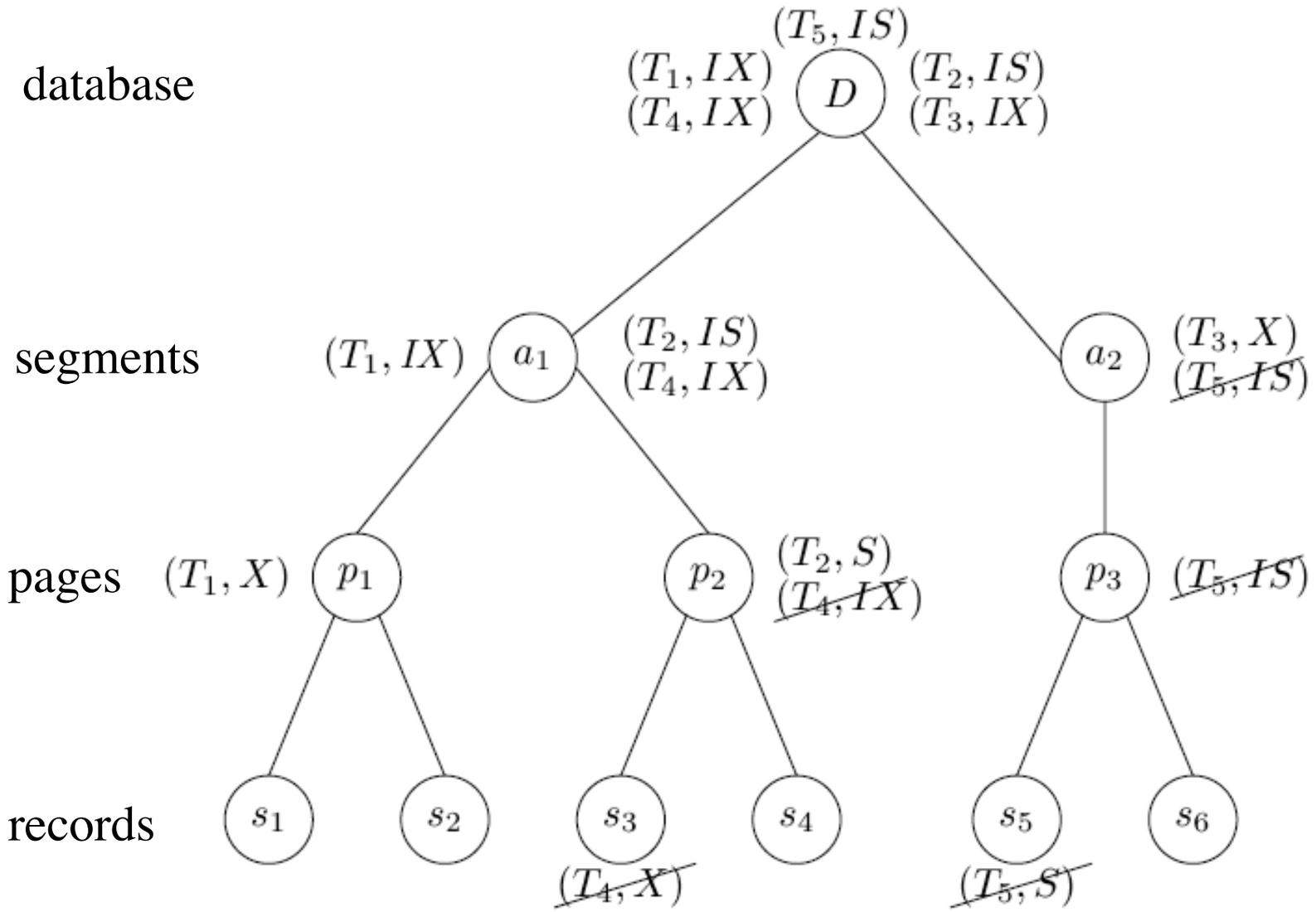
	NL	S	X	IS	IX
S	√	√	–	√	–
X	√	–	–	–	–
IS	√	√	–	√	√
IX	√	–	–	√	√

Example: MGL (1/2)



- T_1 wants to obtain X-lock on page p_1
- T_2 wants to obtain S-lock on page p_2
- T_3 wants to obtain X-lock on segment a_2

Example: MGL (2/2)



- T_4 wants to obtain an X-lock on record s3, T_4 has to wait
- T_5 wants to obtain an S-lock on record s5, T_5 has to wait
- T_4 and T_5 are waiting, however, this is not a deadlock
- Note: MGL may run into deadlocks!

How to handle Insert and Deletes

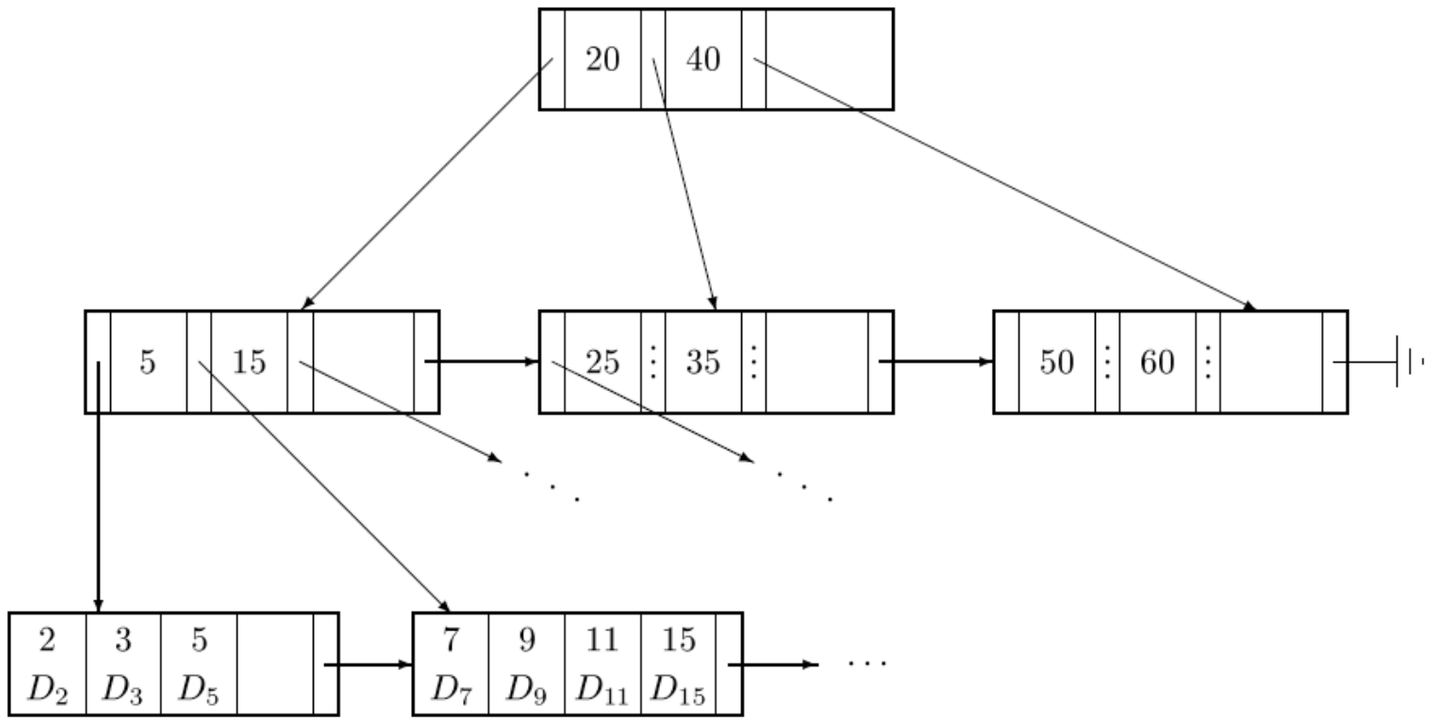
- one approach
  - if a TA wants to delete a data item, it has to obtain an X-lock before deleting
  - any other TA waiting for a lock on the same object will not be able to obtain the lock after the deleting TA successfully committed
  - if a TA wants to insert an item, the TA obtains an X-lock before inserting
- in both cases 2PL is used
- however, this approach does not solve the phantom problem

Phantom Problem

T_1	T_2
select count$(*)$	
from exam	
where score **between** 1 **and** 2;	
	insert into exam
	values(29555, 5001, 2137, 1);
select count$(*)$	
from exam	
where score **between** 1 **and** 2;	

- may be solved by additionally locking the access path to the data items.
- For instance, if a secondary access path exists on “score“, we could
  S-lock the interval [1;2] for T_1.
- Now, if T_2 tries to insert tuple (29555, 5001, 2137, 1) into table *exam*, T_2 has to wait.
- In addition, individual locks have to be obtained on the records as these records may also be accessed without using the secondary access path.

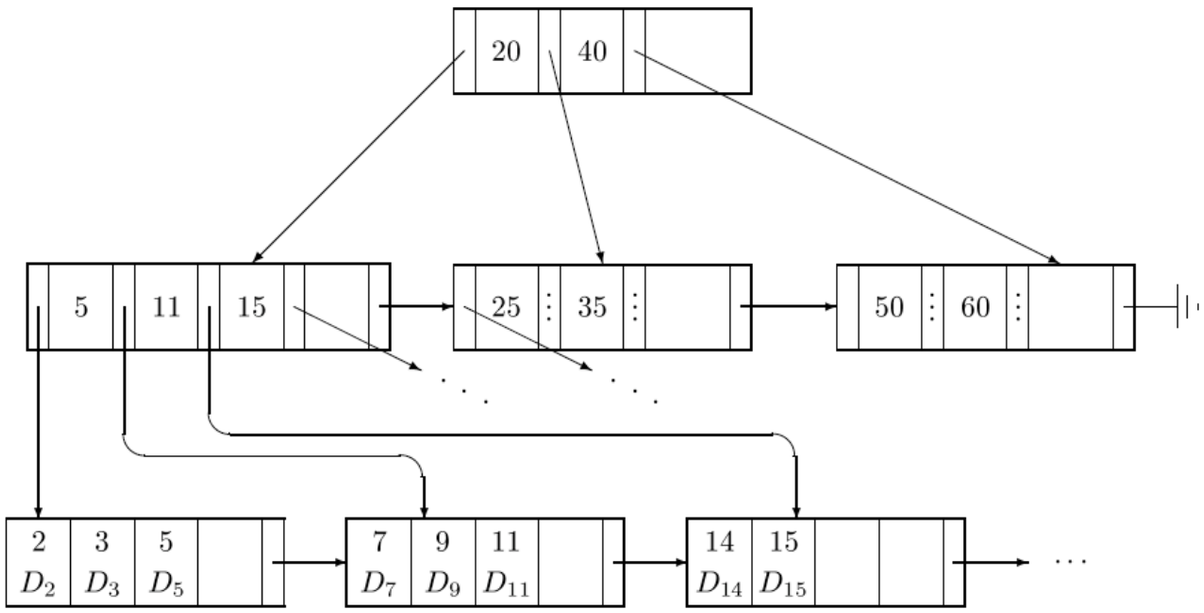
Synchronization of Indexes



- strict 2PL would lock entire path from root to leaf
- Effect: large regions of the B+-tree are locked unnecessarily
- idea: only lock individual nodes rather than the entire path
- required: right-sibling pointers for (non-leaf) nodes

Synchronization of Indexes

- two parallel operations: probe(15) & insert(14)



- probe reaches leaf [7;15] and is suspended
- insert splits leaf [7;15] into [7;11] and [14;15]
- probe continues: key 15 is not on leaf [7;11] anymore!
- probe has to continue searh on the right sbiling
- may also happen on non-leaf nodes

Optimistic Synchronization

Core idea: only when a TA commits, we check whether there was a conflict

One variant of this approach:

1. read phase

- TA performs all operations including change operations
- changed values are not inserted into the database but kept locally for the TA
- all changes are performed on the local variables (other TAs do not see these changes yet)

2. validation phase

- check whether TA might have had a conflict with other TAs
- this will be decided based on timestamps that are assigned to TAs in the order TAs are validated

Optimistic Synchronization

3. Write phase

- if the validation of a TA is positive, its changes will be inserted into the database, otherwise the TA is rolled back

- **Discussion**
  - TAs that are rolled back do not have an impact on the database
  - therefore no cascading rollback may happen
  - the database only contains changes done by successfully committed TAs

Validation Phase

- simplifying assumption: only one TA is validated at a time
- we want to validate transaction T_j.
- the validation is successful if for all transactions T_a - those that already passed the validation phase - one of the following conditions holds:
  - T_a was already finished (including the write phase) when T_j started.
  - the set of data items changed by T_a, termed *WriteSet(T_a),* does not contain any element of the set of data items that were read by T_j, termed *ReadSet(T_j)*. It has to hold:

$$WriteSet(T_a) \cap ReadSet(T_j) = \varnothing$$

Isolation Levels in SQL

- **problem:**
  serializability may considerably degrade overall system performance
- **idea:**
  trade concurrency for consistency in a controlled manner

set transaction
[read only, | read write,]
[**isolation level**
read uncommitted, |
read committed, |
repeatable read, |
serializable,]
[**diagnostics size** . . . ,]

- four different **isolation levels**
- all levels hold long duration **write**-locks (strict 2PL)
- **read uncommitted**
  - reads inconsistent data (uncomitted values, repeatable reads, phantoms)
  - may only be used for read-only TAs
  - useful for browsing a database, e.g., aggregations
  - TA does not slow down execution of other TAs running in parallel
  - no locks are obtained on **read** operations

Isolation Levels in SQL

- **read committed**
  - reads committed values
  - **non-repeatable read**-problem
  - phantom problem still possible
  - TA uses short duration **read** locks on data items
- **repeatable read**
  - non-repeatable read is excluded
  - phantom problem still possible
  - TA uses long duration **read** locks on data items (strict 2PL)
- **serializable**
  - serializability of all TAs
  - should be the default (however, some DBMSs have a different default)
  - TA uses long duration **read** locks on data items (strict 2PL) and predicate locking

T_1	T_2
read(A)	
	write(A)
	write(B)
	commit
read(B)	
read(A)	
...	

Next Topic: Parallelization of Data and Queries.