



Lecture

Distributed Data Management

Chapter 2: Concurrency Control

Erik Buchmann
buchmann@ipd.uka.de



Outline

- Introduction/Problems,
- Definitions
(transaction, history, conflict, equivalence, serializability, ...),
- Locking.





Features of Transactions

- **ACID**

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**

Constraints

Logging,
Recovery

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Atomicity, Isolation

- Transactional guarantees – in particular, **atomicity** and **isolation**.
- Atomicity
 - Example, „bank scenario“:

Bank	Person	Balance
Sparkasse	Klemens	5000
Deutsche Bank	Gunter	200

Which distribution/replication scenarios are possible?

- Money transfer – two elementary operations.
 - debit(Klemens, 500),
 - credit(Gunter, 500).
- Isolation – can be explained with this example, too





Synchronisation, Distribution (1)

- Essential feature of databases:
Many users can access the same data *concurrently* – be it read, be it write.
- Consistency must be guaranteed –
task of **synchronization component**.
- Multi-user mode shall be hidden from users
as far as possible: concurrent processing
of requests shall be transparent,
,illusion' of being the only user.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Synchronisation, Distribution (2)

- **Serial execution** of application programs
 - achieves that illusion without any synchronization effort,
 - database consistency at the end of each program,
 - but extremely long delays and insufficient utilization of resources. (Processor is idle during communication and I/O.)

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Synchronization in General

- Uncontrolled **non-serial execution** leads to problems, notably inconsistency:
 - **lost updates**,
 - **non-repeatable read**, multiple reads of the same data return different values
 - **dirty read**, i.e., reads of uncommitted updates,
 - **phantoms**.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Lost Update

- Program T1 transfers EUR 300,- from Account A to Account B, Program T2 credits 3 % interest rate to Account A.
- Where is the lost update?

Step	T1	T2
1	Read(A, a1)	
2	a1 := a1-300	
3		Read(A, a2)
4		a2 := a2 *1.03
5		Write(A, a2)
6	Write(A, a1)	
7	Read(B, b1)	
8	b1 := b1 + 300	
9	Write(B, b1)	





Dirty Read

- Program T2 credits interest rate based on a value that is not part of a consistent state.
- Where is the dirty read?

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)

Step	T1	T2
1	Read(A, a1)	
2	a1 := a1-300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 * 1.03
6		Write(A, a2)
7		commit
8	Read(B, b1)	
9	...	
10	abort	

Not necessarily
invoked by user.





Non-Repeatable Reads

- Program reads data object more than once and sees modification of another program.

Step	T1	T2
1	Read(A, a1)	
2	a1 := a1-300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 *1.03
6		Write(A, a2)
7	Read(A, a3)	
8	...	

Why is this neither a lost update nor a dirty read?

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Phantom Reads

- Program repeats queries during a transaction, but sees different result sets
- Corresponds to Non-Repeatable Reads on sets instead of values

T1	T2
select count(*) from Bank	insert into Bank (name, account, value) values ('Erik B', 29384, 100.00)
select count(*) from Bank	
...	...





Solution without Transactions

- **Serial execution** of all transactions
 - a new transaction does not start before the preceding transaction did a commit/abort
 - the database is consistent before and after each transaction
 - no efforts for locking, scheduling
 - So why is this not an option for centralized / distributed DBMS?

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Means of Synchronization

- Combining multiple operations to one transaction
 - all-or-nothing execution
- Locking
 - delay operations until the a conflicting transaction has committed/aborted
- Re-arranging the order of parallel transactions
 - try to run transactions in parallel that read/write different data objects
- Aborting of transactions that conflict with others
 - DBMS has to undo all operations of an aborted transaction

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)



Synchronization in existing DBMS

- Example: Isolation levels in Oracle
Read committed is default setting

Note: Lost updates are never possible

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

[Introduction/Problems](#)

[Definitions](#)

[Locking](#)



Definitions



Transactions (1)

- Execution of a program that manipulates the database.
- Representation of the execution that identifies
 - the reads and writes,
 - the order of their execution,
 - whether or not there is a commit at the end (or abort).

[Introduction/
Problems](#)
[Definitions](#)
[Locking](#)





Transactions (2)

- Example:

Procedure P begin

Start;

temp := Read(x);

temp := temp + 1;

Write(x, temp);

Commit

end

operations

- Representation: $r_1[x] \rightarrow w_1[x] \rightarrow c_1$
- Transaction is partial order $(\Sigma, <)$
(Σ will mostly be omitted in what follows.)





Conflict

- Two operations p, q conflict $:=$
 p, q operate on the same data object,
and p or q is a write.
- Further operations –
definition of conflict must be extended.
- What is the compatibility matrix for
read/write/increment/decrement?

	Read	Write	Increment	Decrement
Read				
Write				
Increment				
Decrement				

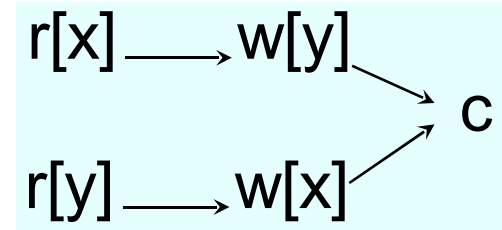
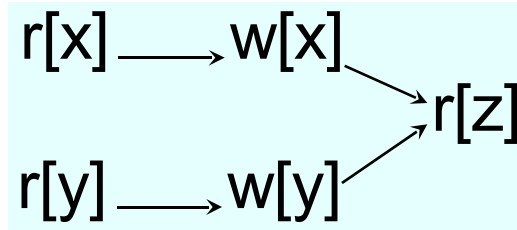
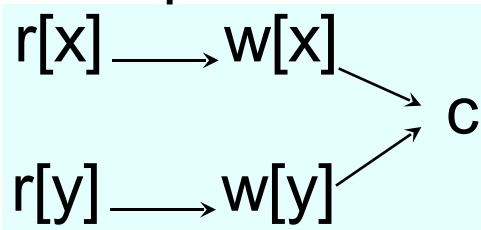


Transaction – Formal Definition

TA is partial order with ordering relation $<$, s.t.

- $T_i \subseteq \{r_i[x], w_i[x] | x \text{ is a data object}\} \cup \{a_i, c_i\}$
- $a_i \in T_i \Leftrightarrow c_i \notin T_i$
- if t is c_i or a_i ,
then for each other operation $p \in T_i$ holds: $p <_i t$
- if $r_i[x], w_i[x] \in T_i$ then $r_i[x] <_i w_i[x]$ or $w_i[x] <_i r_i[x]$

Examples:



Which example is no transaction, and why not?



Reads-from Relationship between Transactions

- *Transaction T_i reads-from transaction T_j in a certain execution if*
 - T_j reads x after T_i has written x , and
 - T_i does not abort before T_i reads x , and
 - each transaction that writes x before T_i reads x and after T_j writes x aborts before T_i reads x .

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Examples for Reads-from Relationships

- $w_1[x] \ w_2[x] \ r_3[x] \ r_4[x] \ c_1 \ c_2 \ c_3 \ c_4$

reads-from relationships:

- $w_1[x] \ w_2[x] \ r_3[x] \ r_4[x] \ c_1 \ a_2 \ c_3 \ c_4$

reads-from relationships:

- $w_1[x] \ w_2[x] \ a_2 \ r_3[x] \ r_4[x] \ c_1 \ c_3 \ c_4$

reads-from relationships:

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Complete Histories

- *Concurrent* execution of the operations of several transactions that are ‘intertangled’ with each other
- Formally:
Let $T = \{T_1, T_2, \dots, T_n\}$ be a set of transactions.

A complete history H over T :=
partial order with order relation $<_H$, such that

- $H = \bigcup_{i=1}^n T_i$
- $<_H \supseteq \bigcup_{i=1}^n <_i$
- $p, q \in H$ have conflict $\Rightarrow p <_H q$ or $q <_H p$





Histories

- Complete histories are defined only after **all** transactions have been committed/aborted.
→ notation is quite useless for scheduling
- In the following, we limit our observations to histories:

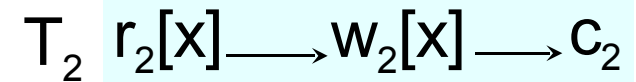
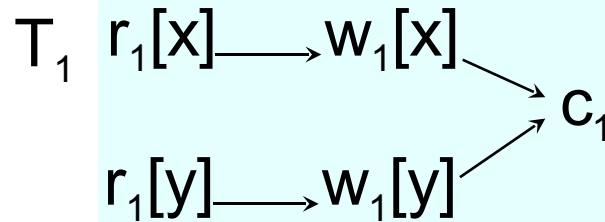
history := prefix of a complete history.



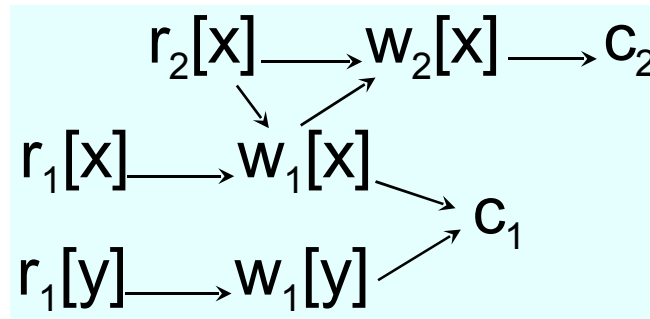


Histories – Examples (1)

- Two transactions:



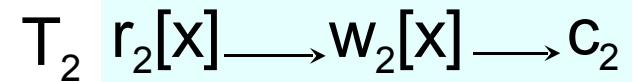
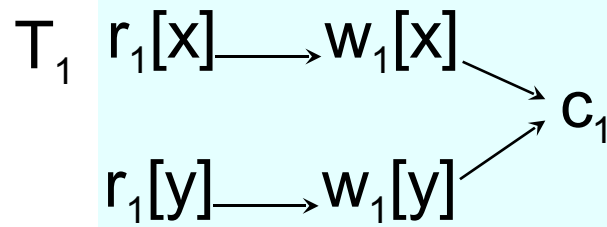
- Complete history:



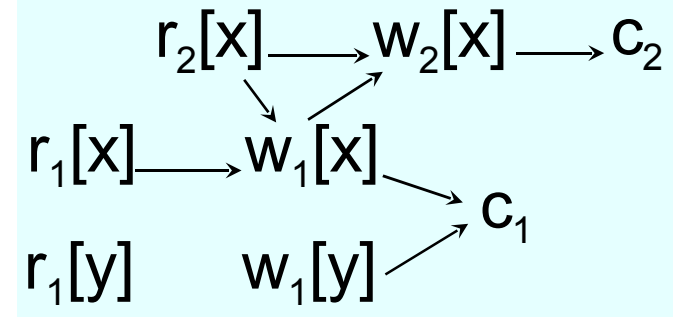
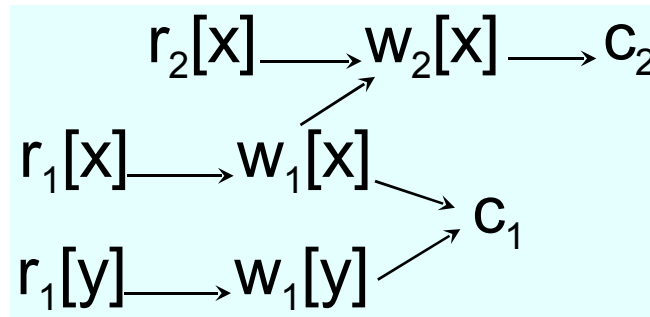


Histories – Examples (2)

- Two transactions:



- No histories:

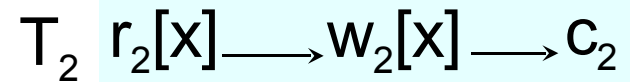
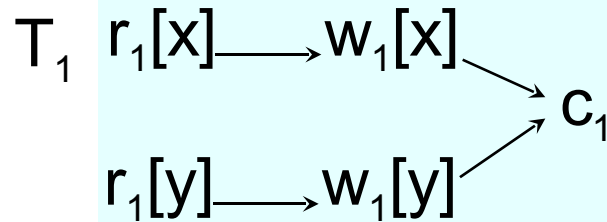


Why not?

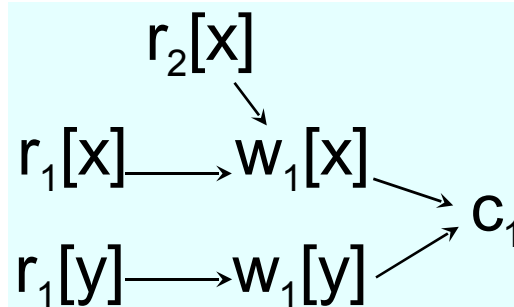


Histories – Examples (3)

- Two transactions:



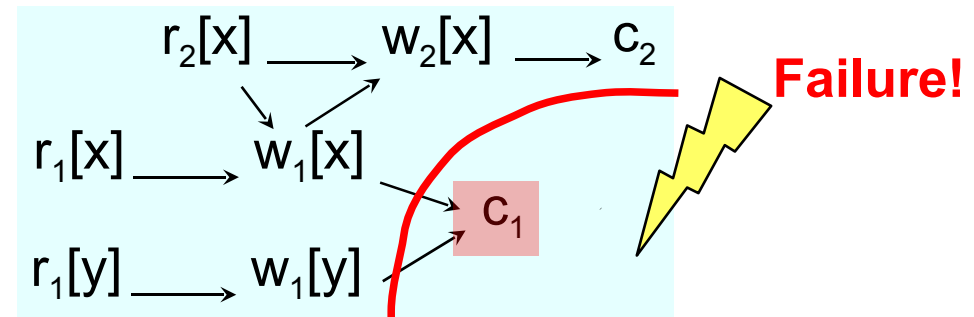
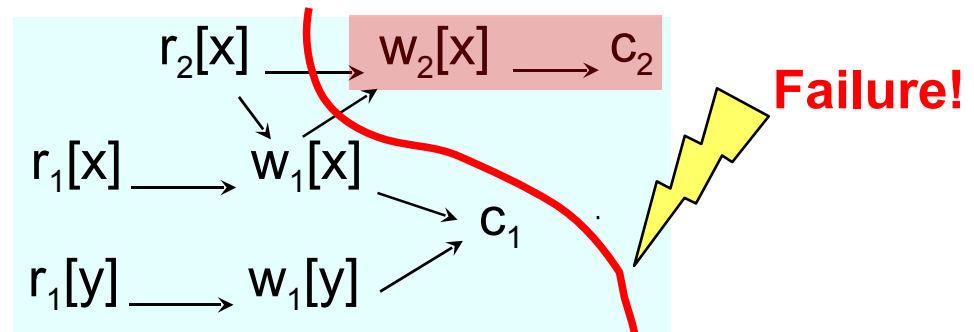
- History, but not a complete history:





Committed Projections

- *Committed projection of history H* –
Abbreviation: $C(H)$:= results from H
by removing all operations that are not committed.
- Illustration:





Correctness of Histories

- History need not be 'correct', e.g., it may contain lost updates
- In order to formally define correctness, we need the definition of **Prefix Commit-Closedness**
 - Characteristic of histories is *prefix commit-closed* if:
characteristic holds for $H \Rightarrow$ characteristic holds for $C(H')$, H' is prefix of H , $C(H) :=$ *committed projection* (only operations from committed TAs)
 - Examples for prefix commit-closed features:
 - history contains less than 17 operations
 - history contains only read-operations
 - history is conflict-free

Introduction/
Problems
[Definitions](#)
[Locking](#)





Equivalence of Histories

- In order to choose among multiple history alternatives, we need a definition for history equivalence
- More than one definition:
 - **conflict equivalence**
 - **view equivalence**

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





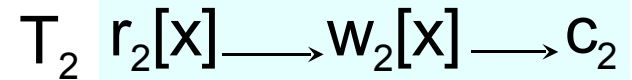
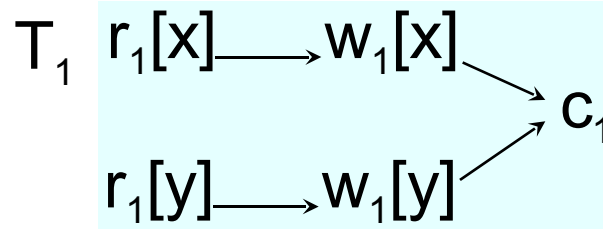
Conflict Equivalence

- Definition 'conflict equivalence':
Histories H, H' are (conflict) equivalent if
 - same transactions, same operations;
 - they establish the same order of operations with conflict.
I.e., p_i, q_j , belong to T_i and T_j , respectively.
 $a_i, a_j \notin H$. If $p_i <_H q_j$ then $p_i <_{H'} q_j$.

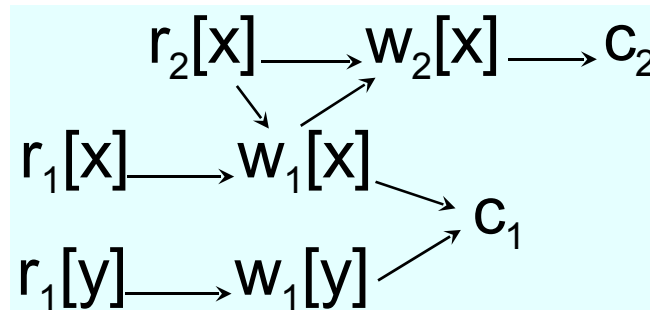


Equivalence of Histories – Examples (1)

- Two transactions:



- Complete history:



- Example of a conflict-equivalent history that is not identical?

Equivalence of Histories – Examples (2)

- History 1:

Step	T1	T2	T3
1	Read(A)	Write(A)	
2			
3	Write(A)		
4			Write(A)

all
transactions
commit

- History 2:

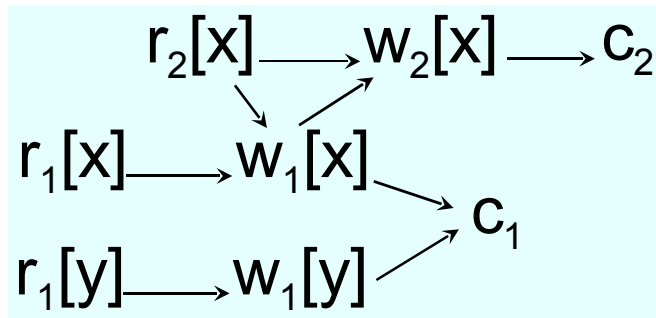
Step	T1	T2	T3
1	Read(A)	Write(A)	
2	Write(A)		
3			
4			Write(A)

- Are these histories conflict-equivalent?



Final Write Operation

- Auxiliary definition – *final write of x in a history*:
operation $w_i[x] \in H$ such that $a_i \notin H$,
and for each $w_j[x] \in H$ ($j \neq i$)
either $w_j[x] < w_i[x]$ or $a_j \in H$.
- What are the final writes of x and y
in the history below?





View Equivalence

- *View equivalence* :=
two histories H, H' are equivalent if
 - same transactions, same operations;
 - T_i, T_j with $a_i, a_j \notin H$ ($a_i, a_j \notin H'$) and for each x :
 T_i reads x from T_j in H .
 $\Rightarrow T_i$ reads x from T_j in H' .
 - For each x : $w_i[x]$ is final write of x in H .
 $\Rightarrow w_i[x]$ is final write of x in H' .

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)



Equivalence of Histories – Examples (3)

- History 1:

Step	T1	T2	T3
1	Read(A)	Write(A)	
2			
3	Write(A)		
4			Write(A)

all
transactions
commit

- History 2:

Step	T1	T2	T3
1	Read(A)	Write(A)	
2	Write(A)		
3			
4			Write(A)

- Are these histories view-equivalent?



Correctness Criteria of Histories

- Conflict-Serializability (**CSR**)
 - enables the database system to generate a serial schedule from the history
- Recoverability (**RC**)
 - allow transactions to abort by recovering the previous database state
- Avoids Cascading Aborts (**ACA**)
 - one aborting transaction does not cause other transactions to abort
- Strictness (**ST**)
 - do not read intermediate results from uncommitted transactions

[Introduction/
Problems](#)

[Definitions](#)

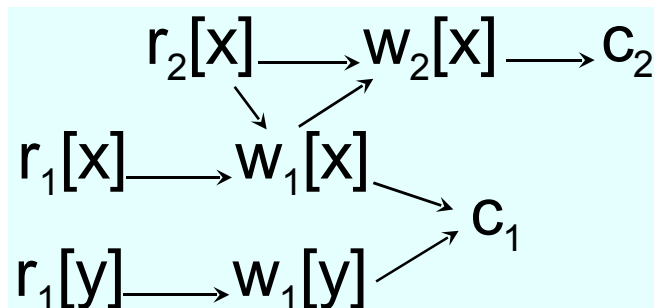
[Locking](#)





Serializability (1)

- Committed projection of history H –
Abbreviation: $C(H) :=$ results from H
by removing all operations that are not committed.
- H is (conflict) serializable
if $C(H)$ is equivalent to serial history H_S .
 - obviously, conflict serializability
is prefix commit-closed.
- Is this history serializable?
If so, what is equivalent serial history?





Serializability (2)

- *H* is view serializable if for each prefix H' $C(H')$ is view equivalent to a serial history.
- Why does this definition include each prefix?
 - view serializability shall be prefix commit-closed
 - Example:
 - $H_{12} = w_1[x] w_2[x] w_2[y] c_2 w_1[y] c_1 w_3[x] w_3[y] c_3$
 $H_{12} = C(H_{12})$
and is view equivalent to $T_1 T_2 T_3$.
 - $H'_{12} = w_1[x] w_2[x] w_2[y] c_2 w_1[y] c_1$
 - According to definition,
 H_{12} is not view serializable.

Introduction/
Problems

Definitions

Locking





Serializability (3)

- H conflict serializable $\Rightarrow$ H view serializable
– but not vice versa.
- Example:
 $H_{13} = w_1[x] \ w_2[x] \ w_2[y] \ c_2 \ w_1[y] \ w_3[x] \ w_3[y] \ c_3 \ w_1[z] \ c_1$

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Recoverability (1)

- commit – DBMS cannot abort it any more.
- Transaction may issue commit, but DBMS decides on point of time when it is executed.
- Commit only when all modifications of data objects that T has read are committed.
- Counterexample:
Dirty Read.

Is it serializable?

Step	T1	T2
1	Read(A, a1)	
2	a1 := a1-300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 *1.03
6		Write(A, a2)
7		commit
8	Read(B, b1)	
9	...	
10	abort	





Recoverability (2)

- *Execution is recoverable* :=
commit of T follows the commit of each
transaction from which T has read.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Cascading Aborts (1)

- Transaction T issues abort operation when it cannot terminate correctly.
- Effects of T must be undone:
 - writes of T,
 - effects of all other transactions that have seen such writes.
- *Undo* may lead to *cascading abort*.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Cascading Aborts (2)

- Example of cascading abort:
 - Initially, x and y both have value 1.
 - Two transactions T_1 and T_2 .
 - Order of operations:
 $\text{Write}_1(x, 2); \text{Read}_2(x); \text{Write}_2(y, 3).$
 - Undo of $\text{Write}_1(x, 2) \Rightarrow T_2$ must abort as well.
- Cascading aborts may happen even though history is recoverable.





Cascadelessness

- Cascading aborts are undesired:
 - they require bookkeeping,
 - Number of transactions that must abort is not bounded.
- *DBMS is cascadeless* :=
Each transaction only reads values of committed transactions.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Strictness

- Undo typically works with *before images*.
- Problem illustration –
original value of x: 1
 - $w_1(x, 2); w_2(x, 3); \text{abort}_1$
 - $w_1(x, 2); w_2(x, 3); \text{abort}_1; \text{abort}_2$
- *strictness* :=
write(x, val) is delayed
until all transactions that have written x before
have committed or aborted,
+ cascadelessness.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Examples

- $T_1 = w_1[x] w_1[y] w_1[z] c_1$
- $T_2 = r_2[u] w_2[x] r_2[y] w_2[y] c_2$
- $H_7 = w_1[x] w_1[y] r_2[u] w_2[x] r_2[y] w_2[y] c_2 w_1[z] c_1$
- $H_8 = w_1[x] w_1[y] r_2[u] w_2[x] r_2[y] w_2[y] w_1[z] c_1 c_2$
- $H_9 = w_1[x] w_1[y] r_2[u] w_2[x] w_1[z] c_1 r_2[y] w_2[y] c_2$
- $H_{10} = w_1[x] w_1[y] r_2[u] w_1[z] c_1 w_2[x] r_2[y] w_2[y] c_2$
- Which histories are recoverable,
which ones can do without cascading aborts,
which ones are strict?

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Serializability Graph (1)

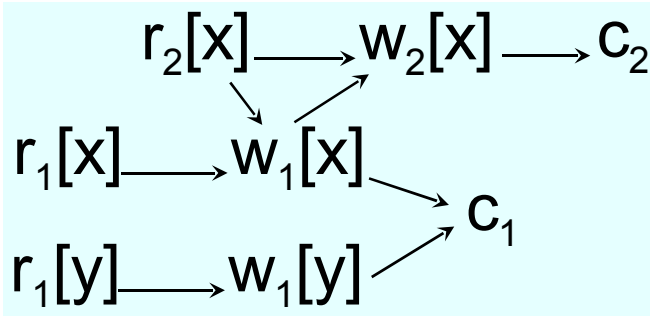
- Delays and resets are common techniques in transaction management.
- Test if certain schedule (= sequence of transactions and their operations) is serializable:
 - generate serializability graph or **dependency graph**.
 - nodes = transactions,
 - edge (directed) = dependency between two transactions:
access to the same data object,
and operations are in conflict.

[Introduction/
Problems](#)
[Definitions](#)
[Locking](#)



Serializability Graph (2)

- Illustration:



What is the serializability graph for this history?



Serializability Graph (3)

- Theorem: schedule is serializable if corresponding dependency graph is *cycle-free*.
- There is a partial order that can be extended to a total one: equivalent serial schedule.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Serializability Graph – Example

- $r(x)/w(x)$ – read-/write access to data object x .

- Schedule: T_1

T_2

T_3

- Dependency graph:

- Serialization order

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)





Serializability Graph – Discussion

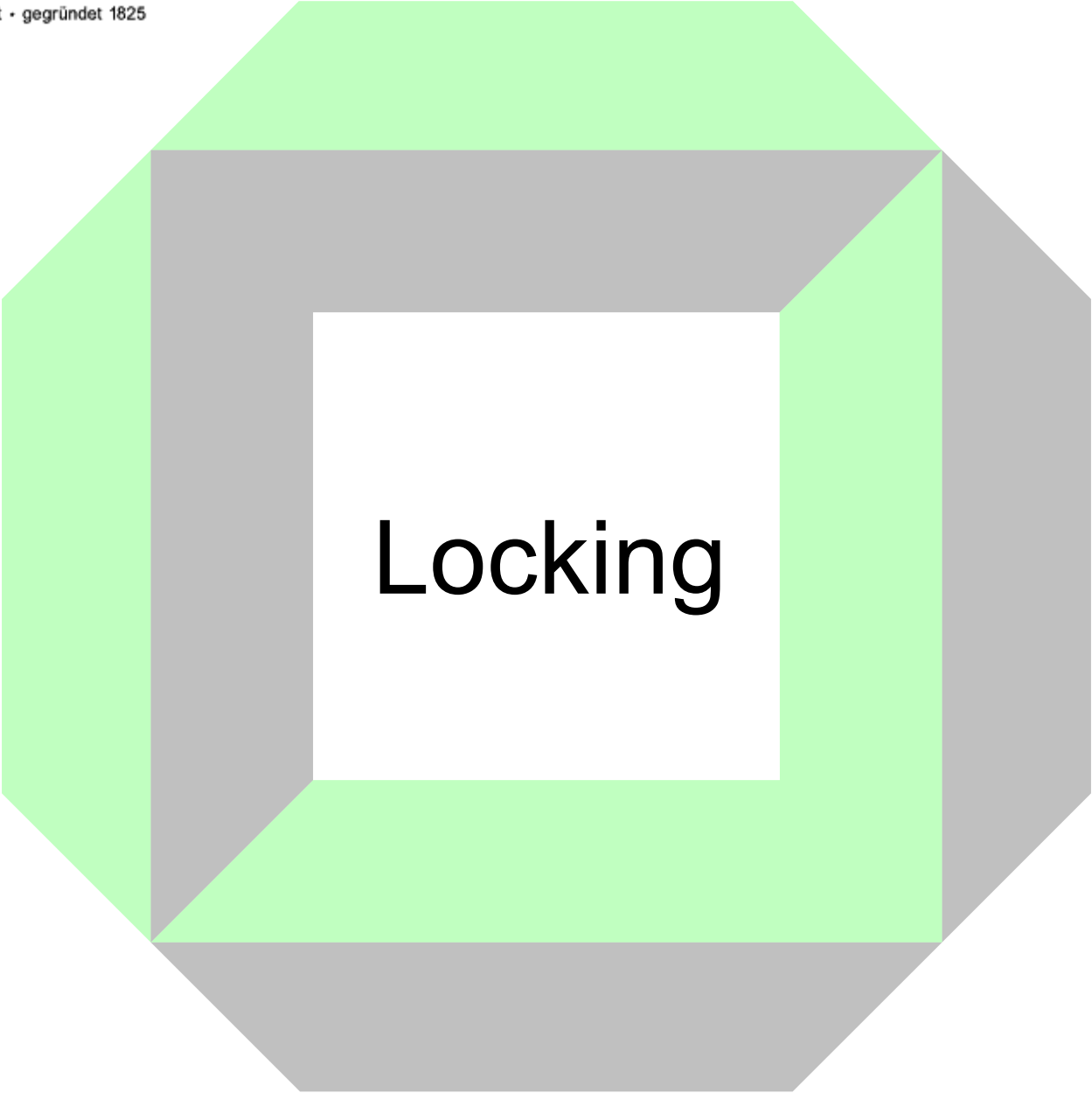
Approach is not practicable.

- Serializability of schedules can only be checked **after** the execution of the transactions.
- Administrative costs are too high: dependencies with terminated transactions need to be taken into account as well.
E.g., relationship between T1 and T3 only becomes clear after commit of T3.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)



Locking



Locking (1)

- Simplest case: only read locks and write locks („RX locking scheme“).
- Lock for each data object and for each operation:
 - read/write lock $rl[x]$, $wl[x]$
 - read/write unlock $ru[x]$, $wu[x]$
- **Centralized locking protocol:**
one designated node handles all lock requests and lock releases.

Introduction/
Problems
Definitions
Locking





Locking (2)

- Transaction obtains lock, transaction releases lock.
 - $wl[x] \rightarrow w[x]$, $rl[x] \rightarrow r[x]$, $ru[x]/wu[x] \rightarrow c$
 - If TA_a wants to lock an object that has been locked by TA_b , it has to wait until ru_b/wu_b
- Locks conflict, just as the operations do
 - multiple $rl[x]$ on the same object x are allowed
 - only one $wl[x]$ is allowed

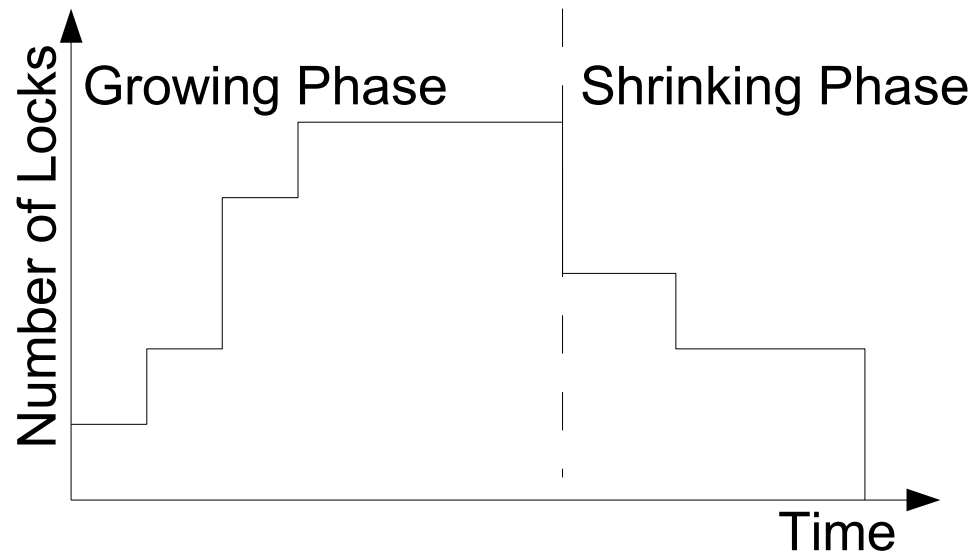
	rl_i	wl_i
rl_j	y	n
wl_j	n	n





Two-Phase Locking

- **Serializability** ensured by two-phase locking protocol (2PL)
 - growing phase
 - shrinking phase



Einleitung/
Probleme
Definitionen
Locking



Example illustrating 2PL

- $T_1: r_1[x] \rightarrow w_1[y] \rightarrow c_1$; $T_2: w_2[x] \rightarrow w_2[y] \rightarrow c_2$

- Without 2PL:

$H_1 = r_1[x] \boxed{r_1[x]} \boxed{ru_1[x]} \boxed{wl_2[x]} \boxed{w_2[x]} \boxed{wl_2[y]} \boxed{w_2[y]} \boxed{wu_2[x]}$
 $\boxed{wu_2[y]} \boxed{c_2} \boxed{wl_1[y]} \boxed{w_1[y]} \boxed{wu_1[y]} \boxed{c_1}$

- $r_1[x] < w_2[x] \wedge w_2[y] < w_1[y]$
 $\Rightarrow \text{SG}(H_1)$ has cycle $T_1 \rightarrow T_2 \rightarrow T_1$

- With 2PL:

$H_1 = r_1[x] \boxed{r_1[x]} \boxed{wl_1[y]} \boxed{w_1[y]} \boxed{c_1} \boxed{ru_1[x]} \boxed{wu_1[y]} \boxed{wl_2[x]} \boxed{w_2[x]}$
 $\boxed{wl_2[y]} \boxed{w_2[y]} \boxed{c_2} \boxed{wu_2[x]} \boxed{wu_2[y]}$



Correctness of 2PL

- Lemmata.
(Let H be the history generated by 2PL scheduler.)
 1. $o_i[x] \in C(H)$
 $\Rightarrow ol_i[x], ou_i[x] \in C(H)$, and $ol_i[x] < o_i[x] < ou_i[x]$.
 2. $p_i[x]$ and $q_j[x]$ ($i \neq j$) conflict.
 $\Rightarrow pu_i[x] < ql_j[x]$ or $qu_j[x] < pl_i[x]$.
 3. $p_i[x], q_i[y] \in C(H) \Rightarrow pl_i[x] < qu_i[y]$.
- Now consider $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$.
- Theorem: each 2PL history H is serializable.



Example for Deadlock with 2PL

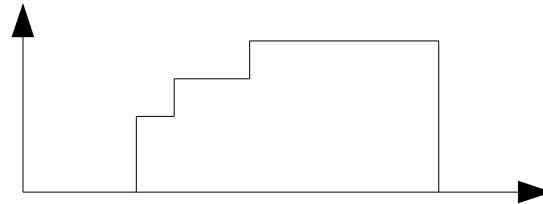
- $T_1: r_1[x] \rightarrow w_1[y] \rightarrow c_1; T_2: w_2[y] \rightarrow w_2[x] \rightarrow c_2$
 1. Both transactions do not have any locks initially.
 2. Scheduler receives $r_1[x]$ from TM. $rl_1[x]$, scheduler submits $r_1[x]$ to DM.
 3. Scheduler receives $w_2[y]$ from TM. $wl_2[y]$, Scheduler submits $w_2[y]$ to DM.
 4. Scheduler receives $w_2[x]$ from TM. $wl_2[x]$ not possible. Delay.
 5. Scheduler receives $w_1[y]$ from TM. $wl_1[y]$ not possible. Delay.

External
reset of
deadlock
required

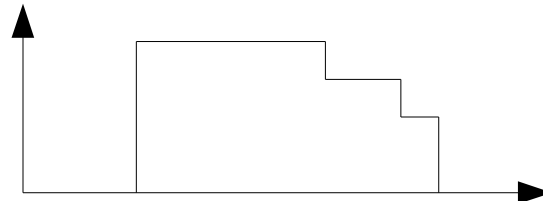


Conservative / Strict 2PL

- Cascadelessness ensured by **strict two-phase locking protocol**: keep all locks until end of transaction



- No deadlocks with the **conservative two-phase locking protocol**: raise all locks at the beginning of transaction



- Combination possible of conservative / strict 2PL





Deadlock – Terminology

- Problem: conservative 2PL causes delays
 - frequently it is more efficient to accept a few deadlocks that can be externally resetted
- But: Which transaction should be resetted?

The following aspects have to be considered:

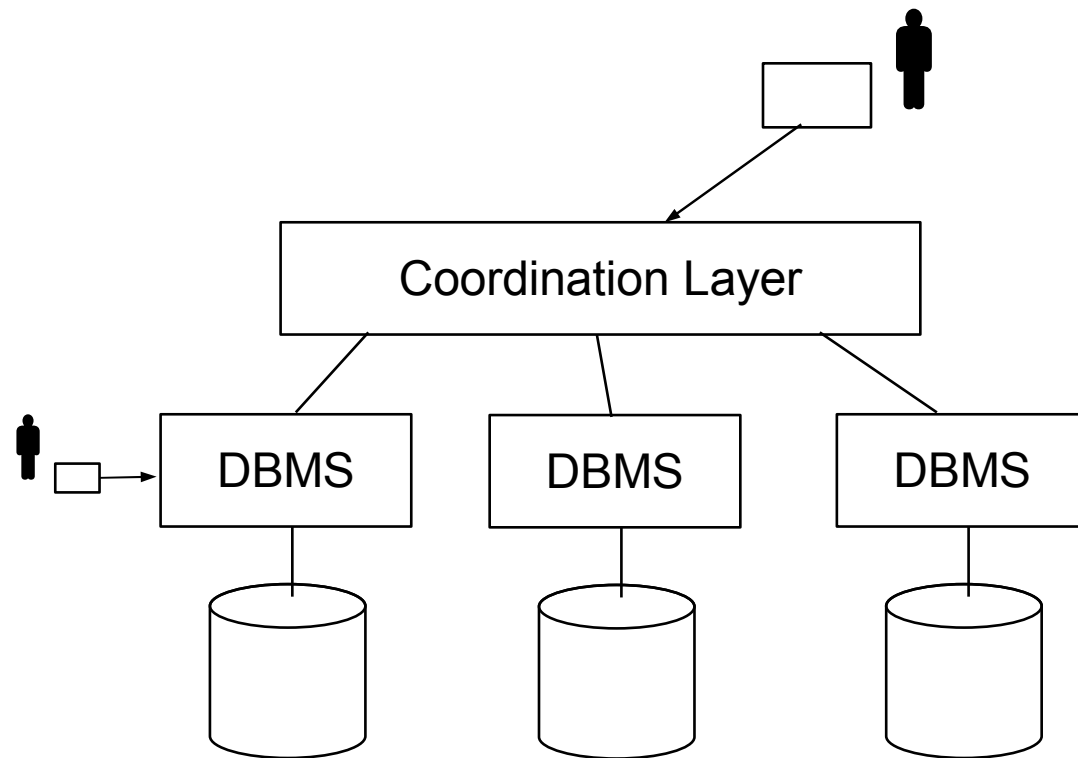
- victim,
- victim selection policy,
- fair victim selection policy.
 - unfair victim selection policy might prevent TA from committing ever



Local vs. Global Transactions

Global
Transactions

Local TAs



- Transaction is global if it refers to more than one database.



Requirements with Distribution

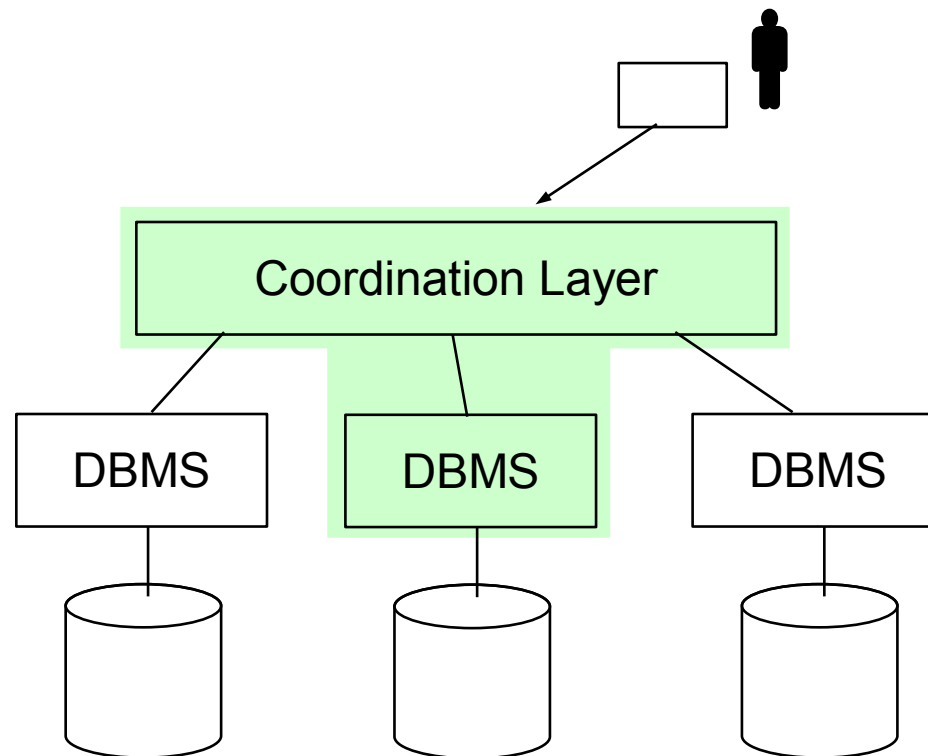
- System-wide serializability of all local and global transactions („global serializability“).
Local serializability (in each node) does not suffice:
Serialization orders may be different. – Example?
- Communication costs should be low.
- As in the centralized case: only as few blockings and resets of transactions as possible.
(These are the methods to deal with synchronization conflicts, negative effect on throughput and answering times.)
- High robustness w.r.t. system faults
(system failure, communication problems).

[Introduction/
Problems](#)
[Definitions](#)
[Locking](#)



Naive Locking in the Distributed Case

- Handle locking as in centralized DBS, i.e., a central synchronization node has total knowledge and locks objects in the databases of other nodes





Discussion

- Advantages:
 - synchronization, deadlock recognition
 - Disadvantages:
 - enormous communication effort required: each lock request becomes a message and its reply, waiting time.
 - synchronization node potential bottleneck for performance and availability („single point of failure“).
 - no autonomy of nodes.
- not really applicable for distributed databases.

[Introduction/
Problems](#)

[Definitions](#)

[Locking](#)



Distributed Locking

- Data is partitioned among nodes.
- Nodes synchronize operations on their partitions
- No further communication (except for replicas): distributed execution of transactions and operations already adapted to distribution of data.
- Release of locks with strict 2PL as part of ACP.
- Biggest problem: global deadlocks.

