Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

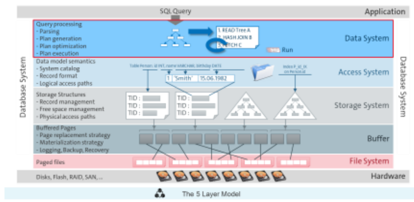
Databases and Information Systems (DIS)

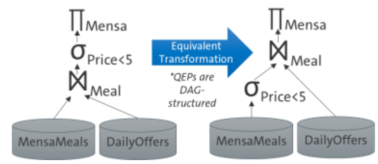
Dr. Annett Ungethüm
Universität Hamburg

Foto: UHH/Esfandiari

Summary

- Architecture of Database Systems
- Transaction Management
- Modern Database Technology
- Data Warehouses and OLAP
- Data Mining
- Big Data Analytics





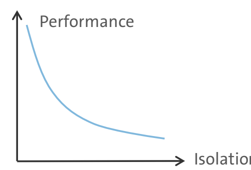
- Architectures of Database Systems
- Transaction Management
- Modern Database Technology
- Data Warehouses and OLAP
- Data Mining
- Big Data Analytics

Atomicity
Consistency and semantic integrity
Isolated execution
Durability

Things that can go wrong, will go wrong: Anomalies

- Dirty Read
- Non-Repeatable Read
- Lost Update
- Phantom Problem

Step	T_1	T_2	T_3	Step	T_1	T_2	T_3
1	r(A)			8		w(C)	
2		r(B)		9		w(A)	
3		r(C)		10			r(A)
4		w(B)		11			r(C)
5	r(B)			12	w(B)		
6	w(B)			13			w(C)
7		r(A)		14			w(A)



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

2

ANSI-SQL isolation levels

Higher isolation / protection against anomalies

Isolation Level	Lost Update possible?	Dirty Read possible?	Non-Repeatable Read possible?	Phantom Read possible?
Read Uncommitted	No	Yes	Yes	Yes
Read Committed	No	No	Yes	Yes
Repeatable Read	No	No	No	Yes
Serializable	No	No	No	No

More concurrency / higher performance

Set isolation level with SQL:
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED| READ COMMITTED |
REPEATABLE READ | SERIALIZABLE }

Special case: read uncommitted → Should avoid Lost Update, but does not always avoid them in reality

A more in-depth discussion and critique of the isolation levels can be found in the following paper:
Berenson, Hal, et al. "A critique of ANSI SQL isolation levels." *ACM SIGMOD Record* 24.2 (1995): 1-10. (https://dl.acm.org/doi/pdf/10.1145/568271.223785)
→ The list of authors might make it look heavily biased, which it might be. But the authors still have a point, and the paper is peer-reviewed

Serializability

The concurrent execution of a set of transactions is considered to be correct, if there is a serial execution of the same set of transactions, leading to **the same resulting DB state** as well as **the same output values** as the original sequential execution.

→ Each schedule that has the same effect as a serial one is considered to be correct
→ A schedule is serializable if there are no loops in its dependency graph

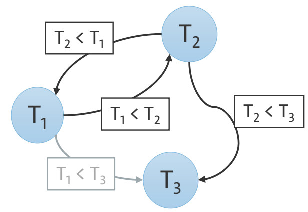
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Complexity of topological sorting to get the possible serializable schedules is ($O(N^2)$)

More about the correctness of schedules:
Vossen, Gottfried. "Database transaction models." *Computer Science Today: Recent Trends and Developments* (2005): 560-574.

Example: Dependency Graph

Step	T_1	T_2	T_3	Step	T_1	T_2	T_3
1	r(A)			8		w(C)	
2		r(B)		9		w(A)	
3		r(C)		10			r(A)
4		w(B)		11			r(C)
5	r(B)			12	w(B)		
6	w(A)			13			w(C)
7		r(A)		14			w(A)



Conflicts

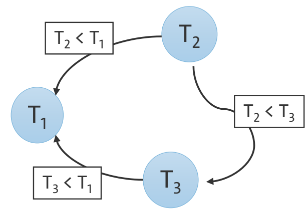
- $w_2(B)<r_1(B)$
- $w_1(A)<r_2(A)$
- $w_2(C)<r_3(C)$
- $w_2(A)<r_3(A)$
- $w_1(A)<w_2(A)$
- $w_2(A)<w_3(A)$
- $w_1(A)<w_3(A)$
- $w_2(B)<w_1(B)$
- $w_2(C)<w_3(C)$

→ Not serializable!

- Cycle: T_1 ←→ T_2
- $w_x(Z) < r_y(Z)$ → Dirty Read
- $w_x(Z) < w_y(Z)$ → Lost Update

Example Schedule (II)

Step	T_1	T_2	T_3
1	r(A)		
2		r(C)	
3		r(B)	
4		w(B)	
5		w(C)	
6			r(C)
7			r(B)
8			w(B)
9	r(B)		

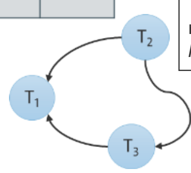


→ Serializable!

- Conflicts:
  - $w_2(B)<r_1(B)$ → DR
  - $w_3(B)<r_1(B)$ → DR
  - $w_2(C)<r_3(C)$ → DR
  - $w_2(B)<r_3(B)$ → DR
  - $w_2(B)<w_3(B)$ → LU

Correct Schedules

Step	T_1	T_2	T_3
1	r(A)		
2		r(C)	
3		r(B)	
4		w(B)	
5		w(C)	
6			r(C)
7			r(B)
8			w(B)
9	r(B)		



A:=1, B:=2, C:=3

T_1	T_2	T_3
read(A)		
	read(A)	
	read(B)	
	B:=B+10	
	C:=5	
	write(B)	
	write(C)	
		read(C)
		read(B)
		B:=B*C
		write(B)
read(B)		

Result: A=1, B=60, C=5

Schedule: T_2, T_3, T_1

A:=1, B:=2, C:=3

T_1	T_2	T_3
	read(C)	
	read(B)	
	B:=B+10	
	C:=5	
	write(B)	
	write(C)	
		read(C)
		read(B)
		B:=B*C
		write(B)
read(A)		
read(B)		

Result: A=1, B=60, C=5

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

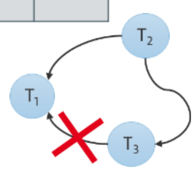
7

Synchronization

- Example for incorrect schedule: T_1, T_3, T_2
  → Results: A=1, B=2*3+10=16≠60, C=5

Correct Schedules (II)

Step	T_1	T_2	T_3
~~1~~	~~r(A)~~		
2		r(A)	
3		r(B)	
4		w(B)	
5		w(A)	
6			r(C)
7			r(B)
8			w(B)
9	r(A)		



Schedule: T_2, T_3, T_1

A:=1, B:=2, C:=3

T_1	T_2	T_3
	read(A)	
	read(B)	
	B:=B+10	
	A:=5	
	write(B)	
	write(A)	
		read(C)
		read(B)
		B:=B*C
		write(B)
read(A)		

Result: A=5, B=36, C=3

Schedule: T_2, T_1, T_3

A:=1, B:=2, C:=3

T_1	T_2	T_3
	read(A)	
	read(B)	
	B:=B+10	
	A:=5	
	write(B)	
	write(A)	
read(A)		
		read(C)
		read(B)
		B:=B*C
		write(B)

Result: A=5, B=36, C=3

Schedules are equivalent!

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

8

Synchronization

- If T_1 does not read *B*, *B* can be changed before or after T_1 has finished

→ T_1 does not require results from T_3 anymore

→ Multiple correct schedules

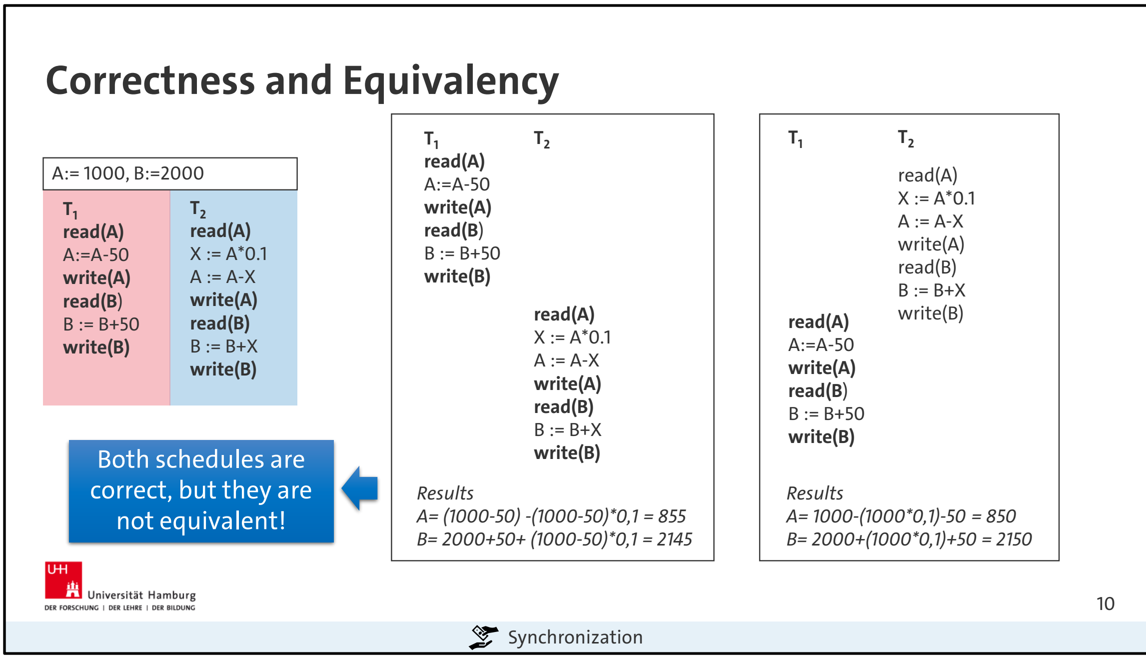
- Effect of a schedule that is not correct:
  - T_3, T_2, T_1 → result: 2*3+10=16≠36

Equivalence

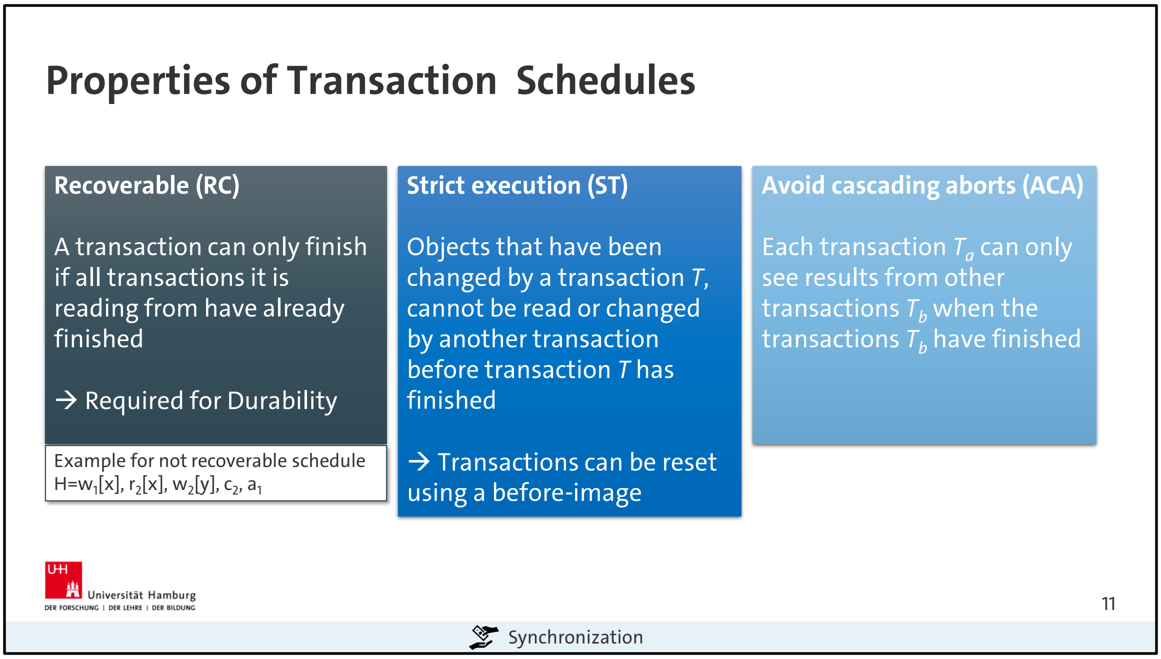
Let $<_H$ and $<_G$ be the ordering relations of the schedules *H* and *G*, then the two execution plans *H* and *G* are considered equivalent if the following conditions are true for all objects *A*, and Transactions *i* and *j*:

$$\text{read}_i[A] <_H \text{write}_j[A] \Leftrightarrow \text{read}_i[A] <_G \text{write}_j[A]$$
$$\text{write}_i[A] <_H \text{read}_j[A] \Leftrightarrow \text{write}_i[A] <_G \text{read}_j[A]$$
$$\text{write}_i[A] <_H \text{write}_j[A] \Leftrightarrow \text{write}_i[A] <_G \text{write}_j[A]$$

→ Not all correct schedules are equivalent!



For n Transactions, there are n! possible serial schedules
→ Reminder from slide 13: Correctness means, that a plan is equivalent to one of these serial schedules, i.e. it has the same effect as one of the serial schedules!



RC

- A schedule is recoverable if the following condition is met:

If a transaction T_i reads from another transaction T_j (i!=j), then $c_j < c_i$

→ The transaction that was read from must commit before the reading transaction

- Example of not recoverable schedule H

→T_2 reads from T_1, but T_2 commits first

ST

- Physical logging using before images

→ Physical logging stores values (instead of operations)

ACA

- If reading results from another TA *T* is allowed before it has finished, an abort can lead to cascading resets

→ All TAs that have read from the aborted TA *T* have to be reset + all TAs that used a result of *T* indirectly

Strict execution avoids cascades, but not necessarily the other way round

(Counter) Examples ACA & ST

Schedule A

Step	T_1	T_2	T_3	T_4	T_5
0.	...				
1.	$w_1[x]$				
2.		$r_2[x]$			
3.		$w_2[y]$			
4.			$r_3[y]$		
5.			$w_3[z]$		
6.				$r_4[z]$	
7.				$w_4[v]$	
8.					$r_5[v]$
9.	a_1				

Schedule B

B = $r_1[x]\ w_1[x]\ w_2[x]\ c_1\ c_2$

Schedule A:
A ∉ ACA → When T_1 is aborted, $T_2 – T_5$ must also be rolled back

Schedule B:
B ∉ ST → $w_2(x)$ takes place before T_1 has finished
<u>But</u>
B ∈ ACA → No transaction reads the result of the other transaction

Database Scheduler

- Sorts the operations and ensures that the schedule/history is serializable and can be rolled back
- Operations can be
  - Executed immediately
  - Rejected
  - On hold
- Strategies:
  - Optimistic
  - Pessimistic

Pessimistic Scheduling

- Operations are set back → Scheduler finds a sequence for operations
- Used when many transactions are expected to change the data
- Most common implementation is based on Locks → Operations are on hold until according lock is released

Optimistic Scheduling

- Scheduler tries to execute operations as soon as possible
- Used When only few transactions are expected to change the data
  - → Damage might happen and must be repaired afterwards
  - → Popular implementations: Time stamp based scheduling, optimistic synchronization

Pessimistic Synchronization

- Locks for exclusive access of data objects

Active lock

Requested lock	NL	R	X
R	+	+	-
X	+	-	-

NL – No Lock
R – Read Lock
X – eXclusive Lock

- Realizes a logical single-user setup
- For each data object, a central lock table stores the lock mode
- R lock: Object can only be read by other transactions, it cannot be changed
- X lock: Object can neither be read nor changed by another transaction
- Example: If an object is already locked with a read lock (R), an exclusive lock (X) cannot be acquired for that object, but another read lock (R) can be acquired.
- Read Lock is also sometimes referred to as “Shared lock (S)”

Working with Locks

Static locks	Dynamic locks
Claimed at the start of a transaction (Preclaiming)	Claimed when needed →Possibility of deadlocks

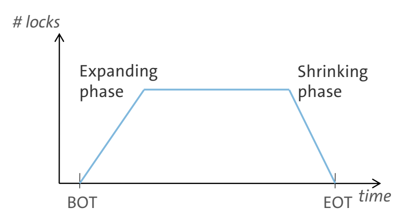
- Locks have to be kept until the transaction finishes to guarantee serializability

→ This constraint can be relaxed for optimization reasons, e.g. early release of read locks

- Remember: We do not always need perfect serializability
- An early release of read locks can lead to different results
- More rules for working with locks:
    - Each object that is used by a transaction, has to be locked before its usage
    - A transaction that is currently holding a lock, cannot claim the same lock again
    - If a lock cannot be claimed (see last 14), the transaction is queued
    - A transaction must not acquire another lock after the first lock has been released (→2-phase-locking-protocol)
    - When a transaction ends, it must release all locks

2-Phase-Locking-Protocol (2PL)

- Expanding phase: Locks are acquired but not released
- Shrinking phase: Locks are released but not claimed



- Locks can only start to be released after all locks have been acquired
- 2PL is almost(!) sufficient for a correct execution of transactions

→ Deadlocks can still happen

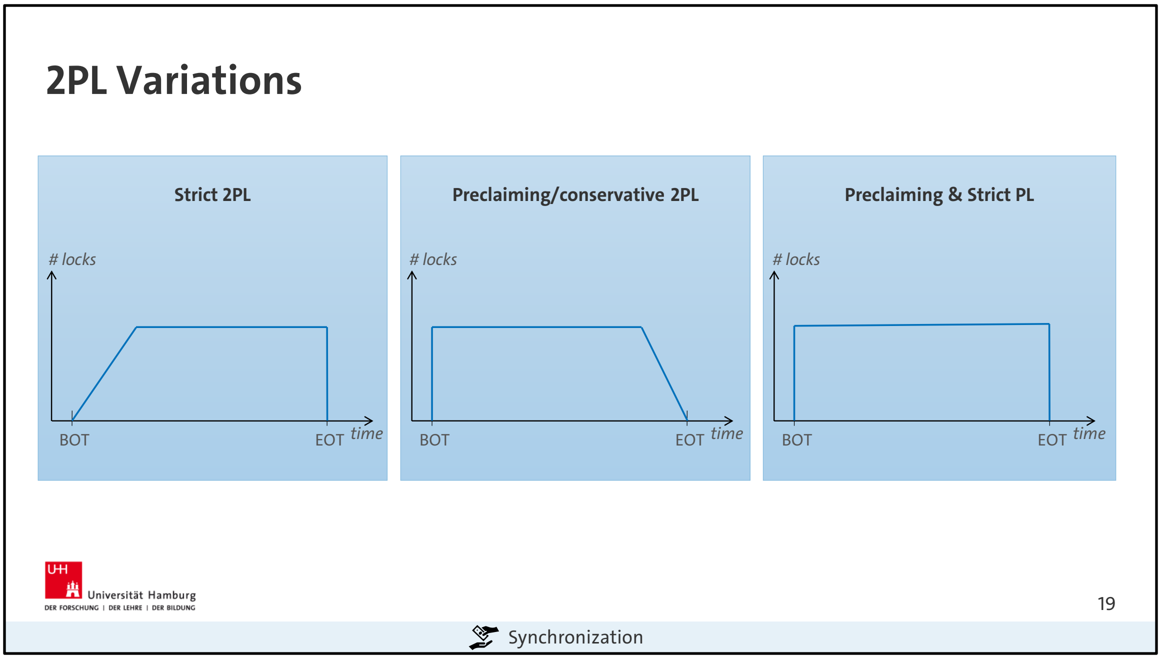
2PL Example

Step	T_1	T_2	Comment
1.	BOT		
2.	lockX[x]		
3.	r[x]		
4.	w[x]		
5.		BOT	
6.		lockR[x]	T_2 must wait
7.	lockX[y]		
8.	r[y]		
9.	unlockX[x]		Wake up T_2
10.		r[x]	
11.		lockR[y]	T_2 must wait
12.	w[y]		
13.	unlockX[y]		Wake up T_2
14.		r[y]	
15.	commit		
16.		unlockR[x]	
17.		unlockR[y]	
18.		commit	

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Expanding phase of T_1: Step 1 to 7
Expanding phase of T_2: Step 5 to 11
Shrinking phase of T_1: Step 9 to 15
Shrinking phase of T_2: Step 16 to 18

Step 6: T_2 cannot aquire lock for x because T1 holds an exclusive lock on x → T_2 must wait
Step 9: The lock of x is released in T_1, T_2 can continue



Strict 2PL

All locks are released after all other operations have finished

- Avoids cascading aborts of transactions that read the results of a rolled back transaction

Preclaiming

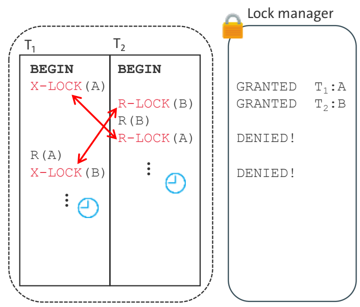
All locks are set before any other operations are executed

- Avoids deadlocks
- Decreases parallelism
- Requires to know all necessary locks at the BOT (not trivial in statements with branches)

Strict 2PL and Preclaiming

Enforces sequential execution

Deadlocks



Next week

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG