Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Databases and Information Systems (DIS)

Dr. Annett Ungethüm
Universität Hamburg

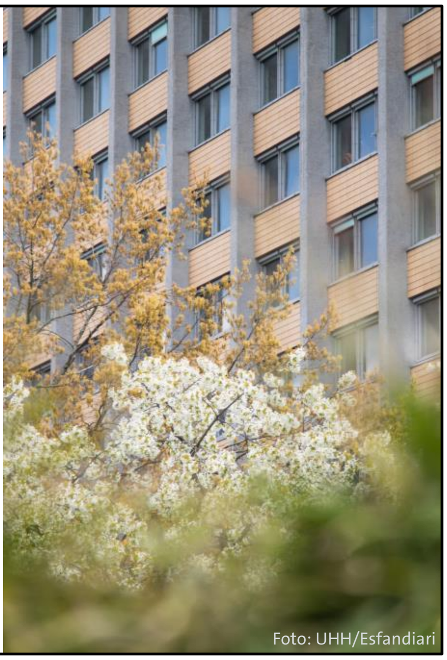
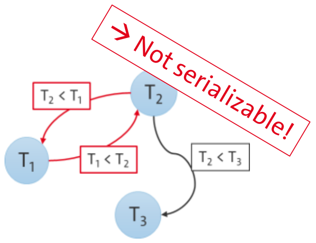
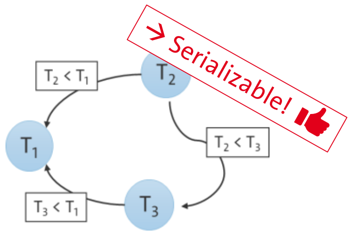


Foto: UHH/Esfandiari

Summary

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





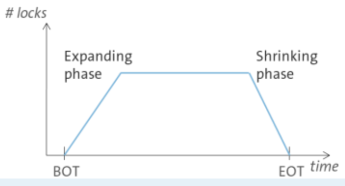
Equivalency

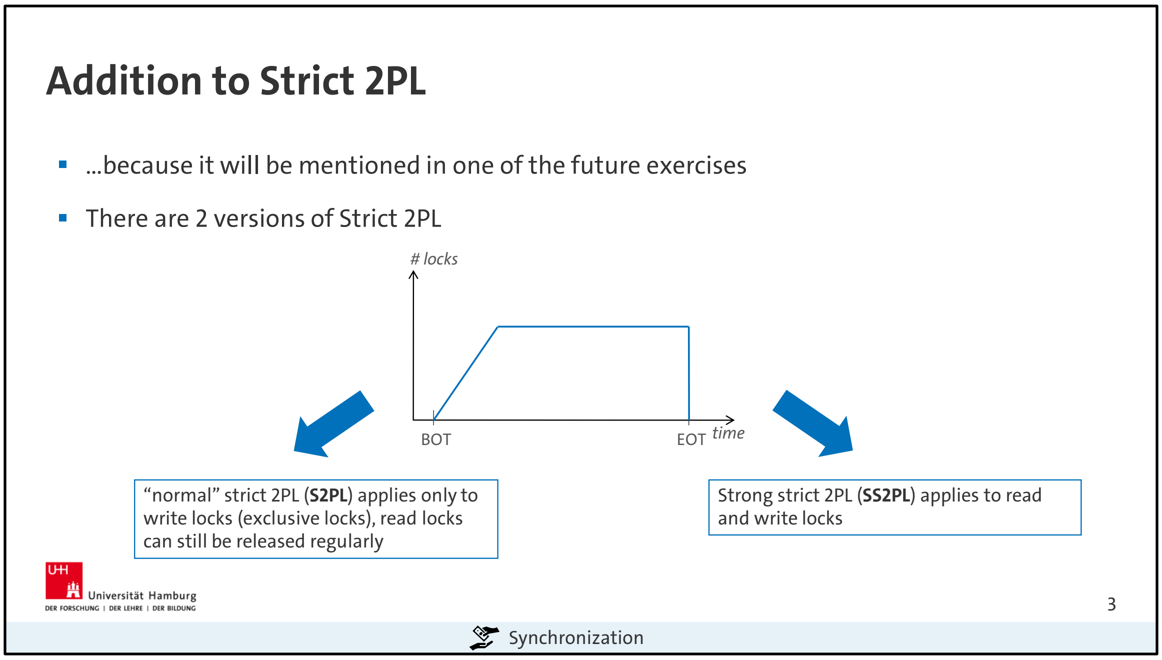
$read_i[A] <_H write_j[A] \Leftrightarrow read_i[A] <_G write_j[A]$
$write_i[A] <_H read_j[A] \Leftrightarrow write_i[A] <_G read_j[A]$
$write_i[A] <_H write_j[A] \Leftrightarrow write_i[A] <_G write_j[A]$

Recoverable (RC)
Strict execution (ST)
Avoid cascading aborts (ACA)

Active lock

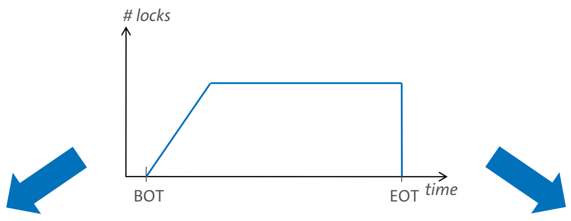
Requested lock	NL	R	X
R	+	+	-
X	+	-	-





Addition to Strict 2PL

- ...because it will be mentioned in one of the future exercises
- There are 2 versions of Strict 2PL

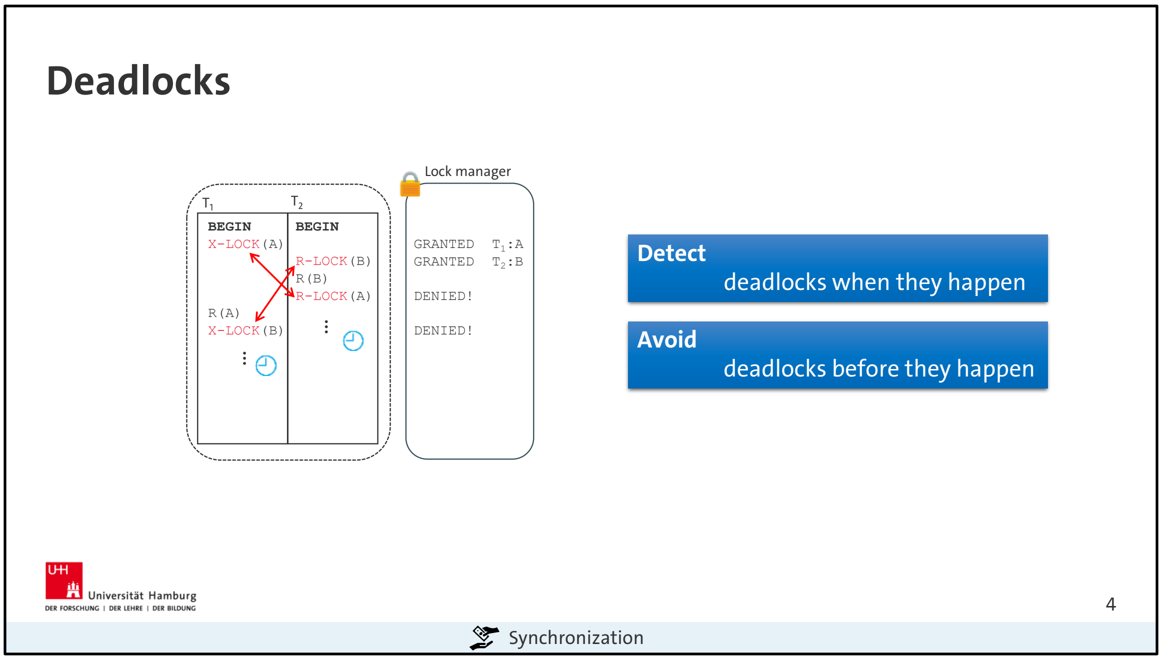


"normal" strict 2PL (**S2PL**) applies only to write locks (exclusive locks), read locks can still be released regularly

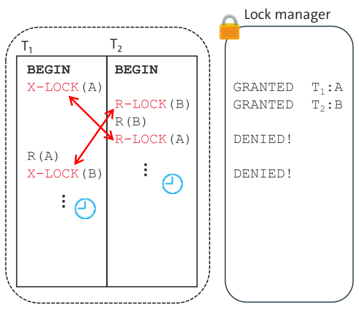
Strong strict 2PL (**SS2PL**) applies to read and write locks

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

- SS2PL schedules are always also S2PL schedules
- SS2PL avoids cascading aborts (ACA)
- SS2PL is sometimes called Rigorous 2PL



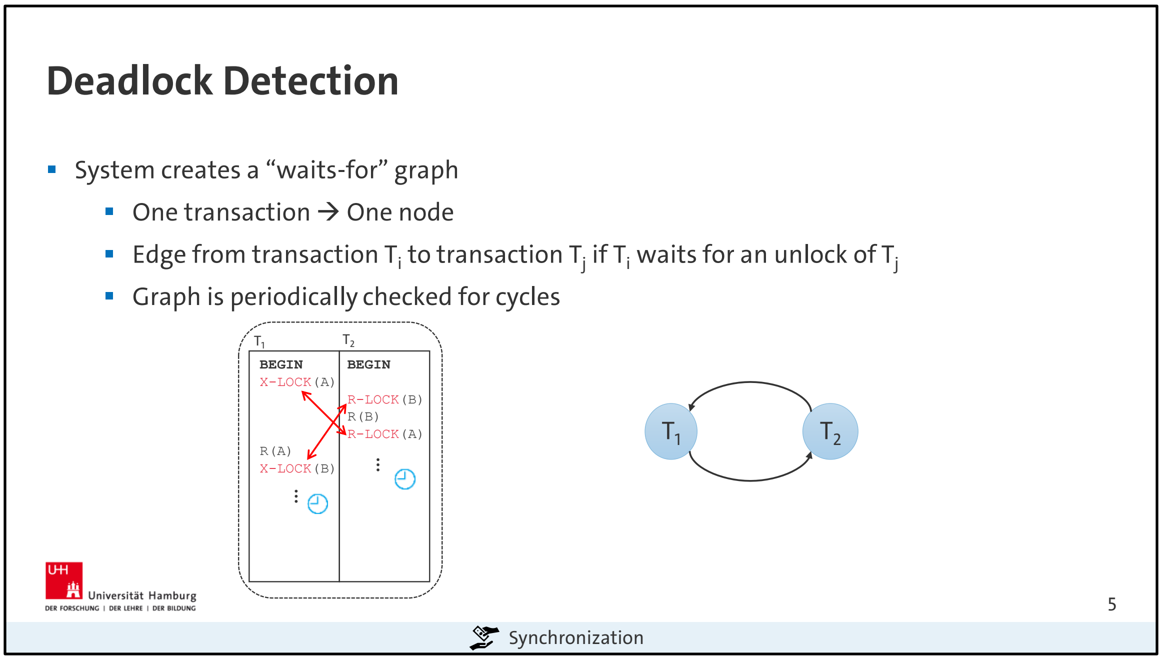
Deadlocks



Detect deadlocks when they happen

Avoid deadlocks before they happen

- Deadlocks cannot be guaranteed to be avoided when locks are used in pessimistic scheduling
- Transactions wait reciprocally for the release of locks
- Requirements for Deadlocks:
    - Parallel access
    - Claim of exclusive locks
    - Claiming transaction already has a lock on data objects
    - No early release of locks (non-preemption → compare with [non]-preemptive scheduling for CPUs by operating systems)



Deadlock Detection

- System creates a “waits-for” graph
  - One transaction → One node
  - Edge from transaction T_i to transaction T_j if T_i waits for an unlock of T_j
  - Graph is periodically checked for cycles

```
T1              T2
BEGIN           BEGIN
X-LOCK(A)
                R-LOCK(B)
                R(B)
                R-LOCK(A)
R(A)                ⋮
X-LOCK(B)
    ⋮
```

$$\begin{tikzcd} T_1 \arrow[r, bend right] & T_2 \arrow[l, bend right] \end{tikzcd}$$

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

5

Synchronization

What to do when a deadlock has been detected?
→Choose one of the transactions and roll it back
→Which transaction to choose? Some heuristics:

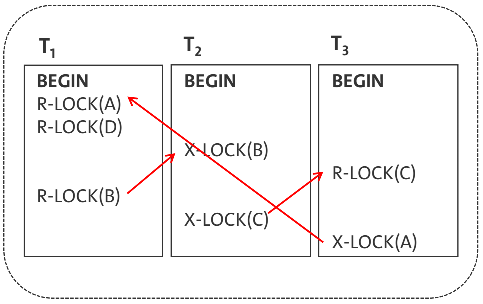
- Most recent time stamp
- Process (smallest number of processed operations)
- Number of held locks
- Number of transactions reading from this transaction
- Number of previous rollbacks on the same transaction (so the transaction is not rolled back every time and can finally finish)

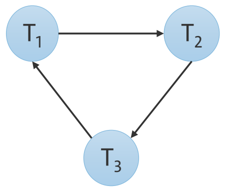
Exercise: Waits-For Graph

T_1	T_2	T_3
BEGIN	**BEGIN**	**BEGIN**
R-LOCK(A)		
R-LOCK(D)		
	X-LOCK(B)	
		R-LOCK(C)
R-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

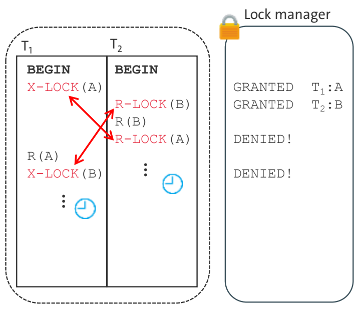
Exercise: Waits-For Graph




$$\begin{tikzcd} T_1 \arrow[rr] & & T_2 \arrow[dl] \\ & T_3 \arrow[ul] & \end{tikzcd}$$

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Deadlocks



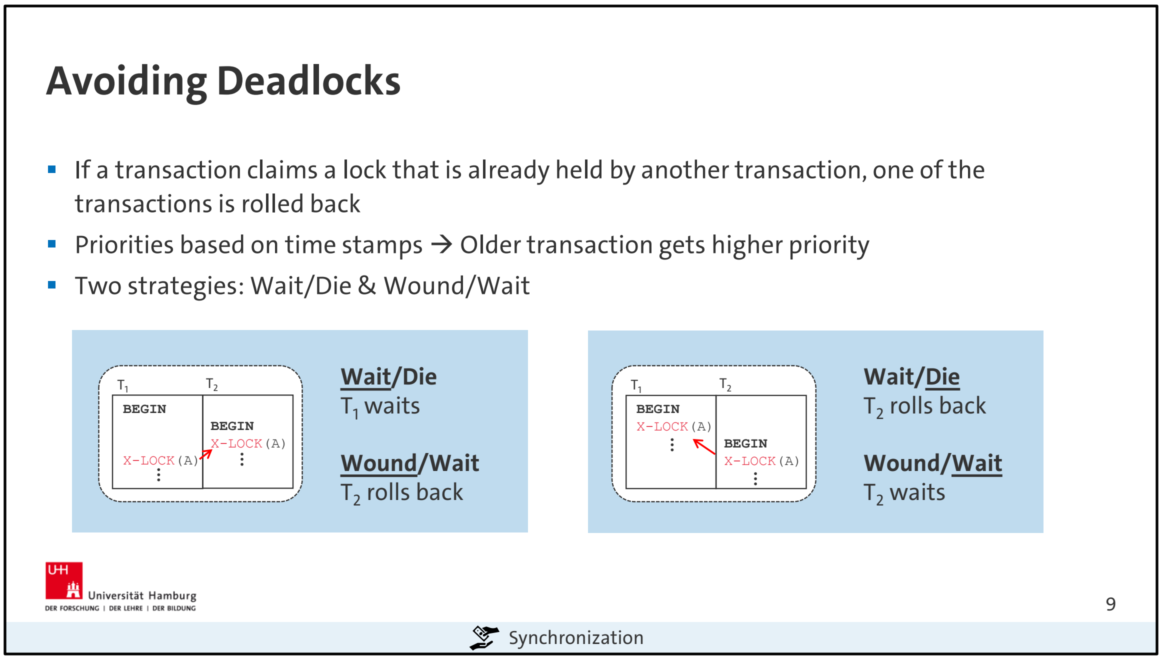
```
BEGIN          BEGIN
X-LOCK(A)
               R-LOCK(B)
               R(B)
               R-LOCK(A)
R(A)
X-LOCK(B)
```

```
GRANTED   T1:A
GRANTED   T2:B

DENIED!

DENIED!
```

Detect deadlocks when they happen

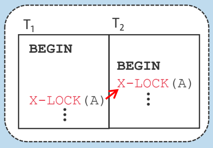
Avoid deadlocks before they happen

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG



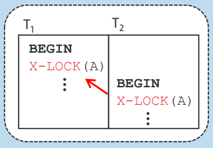
Avoiding Deadlocks

- If a transaction claims a lock that is already held by another transaction, one of the transactions is rolled back
- Priorities based on time stamps → Older transaction gets higher priority
- Two strategies: Wait/Die & Wound/Wait



Wait/Die
T_1 waits

Wound/Wait
T_2 rolls back



Wait/**Die**
T_2 rolls back

Wound/**Wait**
T_2 waits

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

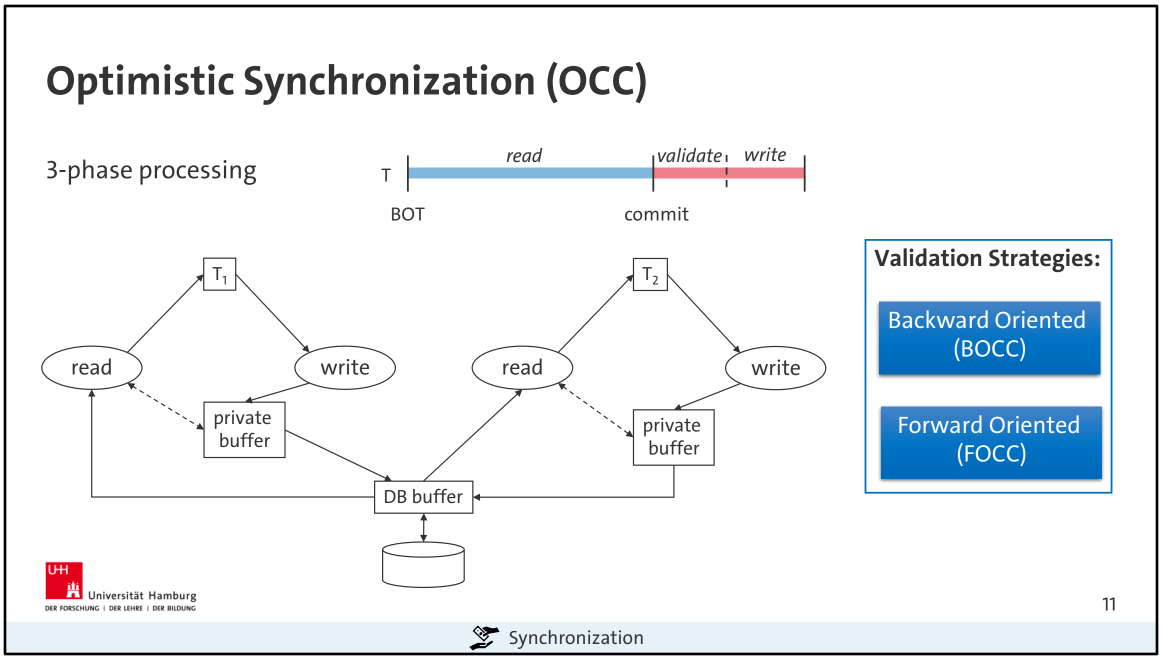
Terminology:

- Name shows what the claiming transaction does if there is a lock conflict
  - Wait: The claiming transaction waits for the lock to be released
  - Wound: Claiming transaction "wounds" the transaction holding the lock, i.e. it tries to roll back the owner of the lock.
  - Die: The claiming transaction is rolled back
- First term: Claiming transaction is older than the transaction holding the lock
- Second term: Claiming transaction is younger than the transaction holding the lock

→ Wait/Die: Claiming transaction waits if it is older than the transaction holding the lock. It rolls back if it is younger than the transaction holding the lock.
→ Wound/Wait: Claiming transaction tries to roll back the transaction holding the lock if the claiming transaction is older. Claiming transaction waits if it is younger than the transaction holding the lock.

Optimistic Synchronization

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG



Optimistic Synchronization (OCC)

3-phase processing

Validation Strategies:

Backward Oriented (BOCC)

Forward Oriented (FOCC)

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

11

Synchronization

- Also called “Optimistic concurrency control” (OCC)
- Used when only few conflicts are expected
- No deadlocks
- Easy reset of transactions
- Enables higher degree of parallelism than pessimistic locking
- Keeps track of the read set (RS) and write set (WS) of each transaction for the validation phase
  - RS: Elements a transaction reads
  - WS: Elements a transaction writes

Read phase

- Executes the transaction
- Changes are stored in a private buffer

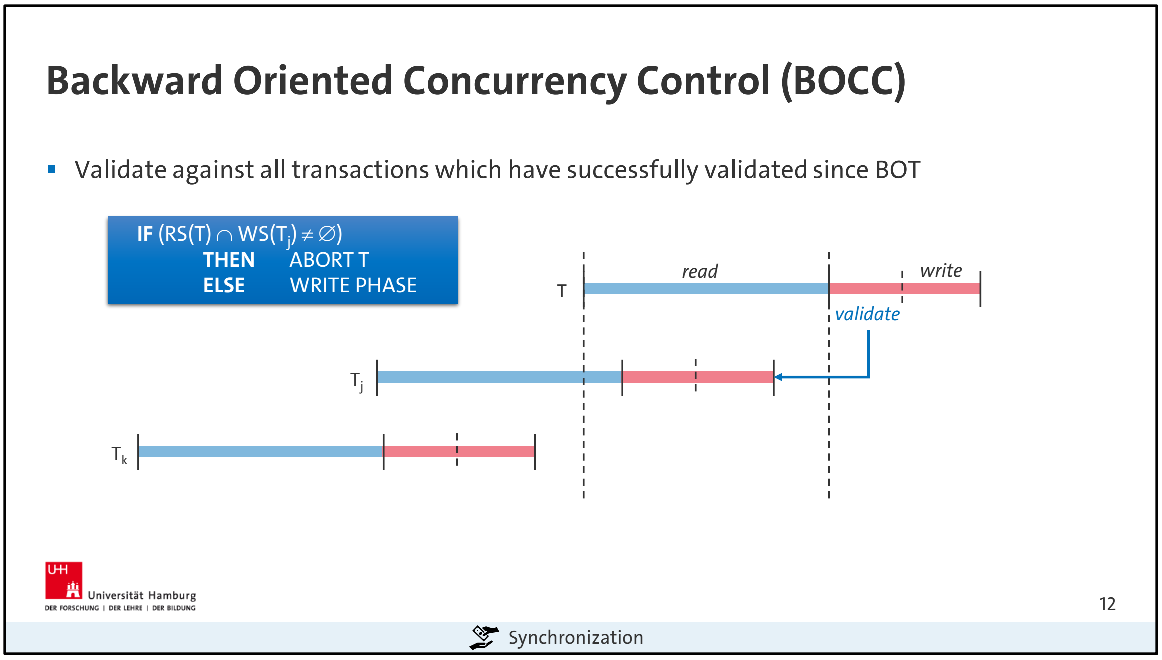
Validate phase

- Check if there are conflicts between parallel transactions (read/write or write/write)
- Resolve conflicts by rolling back transactions
- A transaction can only validate successfully if all changes it has read were from validated transactions

$\rightarrow$ Validation sequence defines sequence of serialization

Write phase

- Only when validation is successful
- Reading transactions are done without additional work
- Writing transaction adds log information and propagates the changes (from private buffer to DB)



Backward Oriented Concurrency Control (BOCC)

- Validate against all transactions which have successfully validated since BOT

```
IF (RS(T) ∩ WS(T_j) ≠ ∅)
    THEN    ABORT T
    ELSE    WRITE PHASE
```

read
write
T
validate
T_j
T_k

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

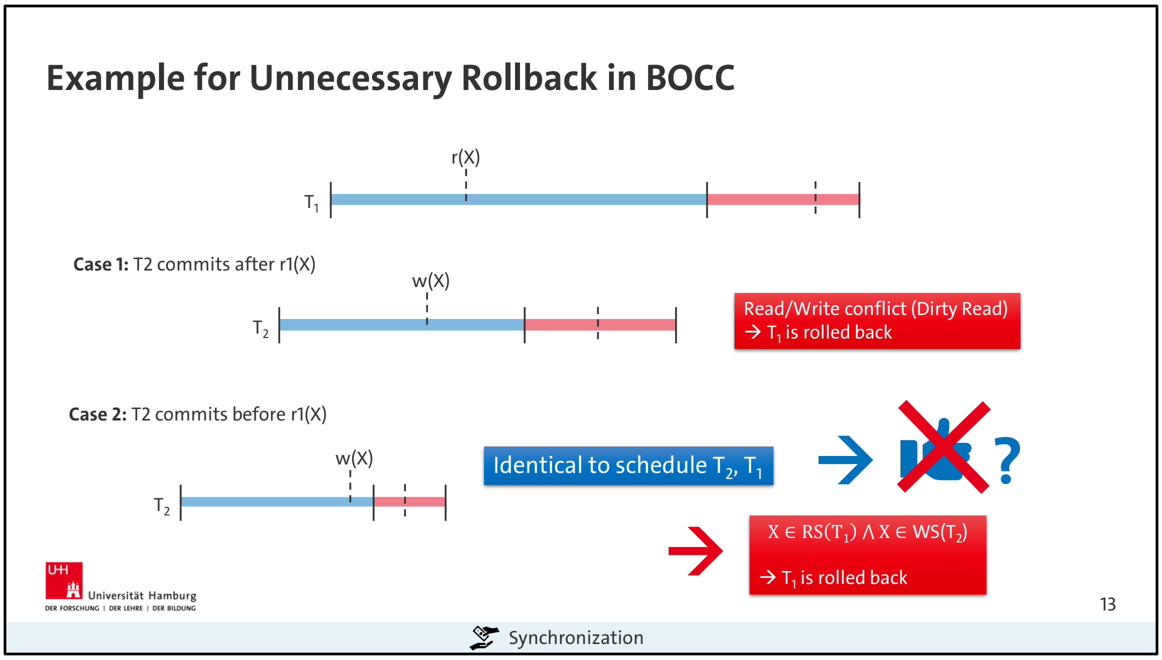
12

Synchronization

- T validates against T_j because T_j successfully validated while T was in the read phase
- T does not validate against T_k because T_k was successfully validated before T started

Disadvantages

- High number of comparisons in validation phase
- High risk of long running transactions to be rolled back
- Roll back only at EOT → Unnecessary work was done
- Requires to keep the WS of already finished transactions
- Can lead to unnecessary rollbacks because of faulty conflict detection



Example for Unnecessary Rollback in BOCC

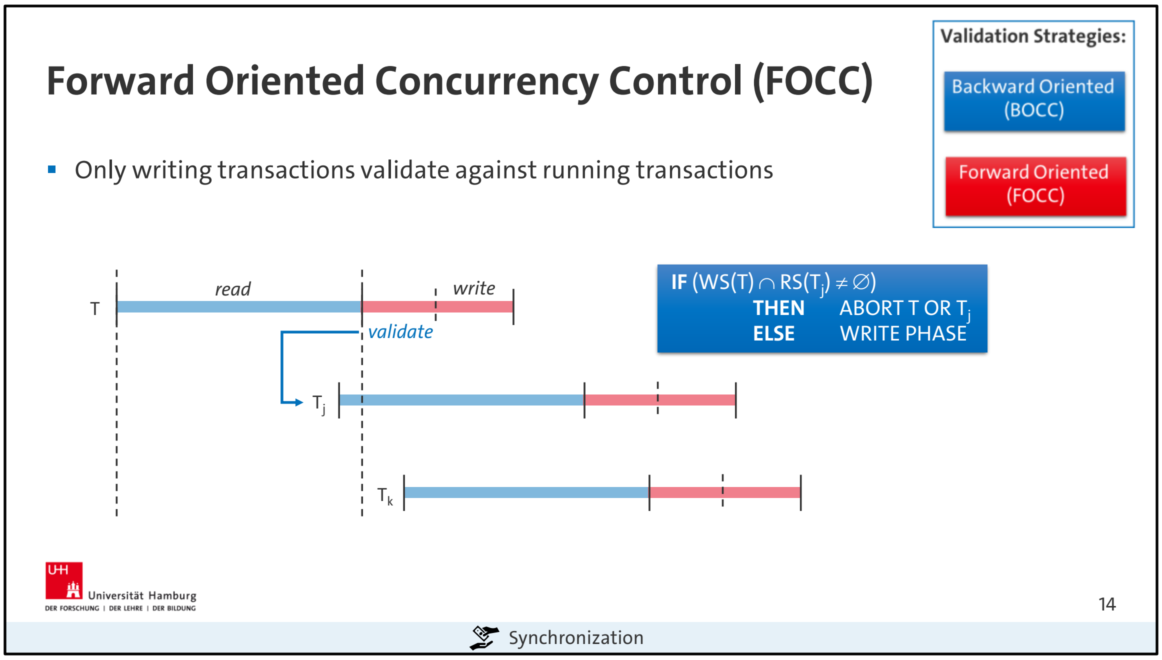
r(X)
T_1
Case 1: T2 commits after r1(X)
w(X)
T_2
Read/Write conflict (Dirty Read)
→ T_1 is rolled back
Case 2: T2 commits before r1(X)
w(X)
T_2
Identical to schedule T_2, T_1
?
$X \in RS(T_1) \land X \in WS(T_2)$
→ T_1 is rolled back
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG
13
Synchronization

Variation: BOCC+

X gets a time stamp

➔ BOCC+ can decide whether X is accessed before or after the commit of T_2

➔ T_1 would not have to be rolled back in case 2 with BOCC+



Forward Oriented Concurrency Control (FOCC)

Validation Strategies:

Backward Oriented (BOCC)

Forward Oriented (FOCC)

- Only writing transactions validate against running transactions

T
read
write
validate
T_j
T_k

IF $(WS(T) \cap RS(T_j) \neq \emptyset)$
THEN ABORT T OR T_j
ELSE WRITE PHASE

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

14

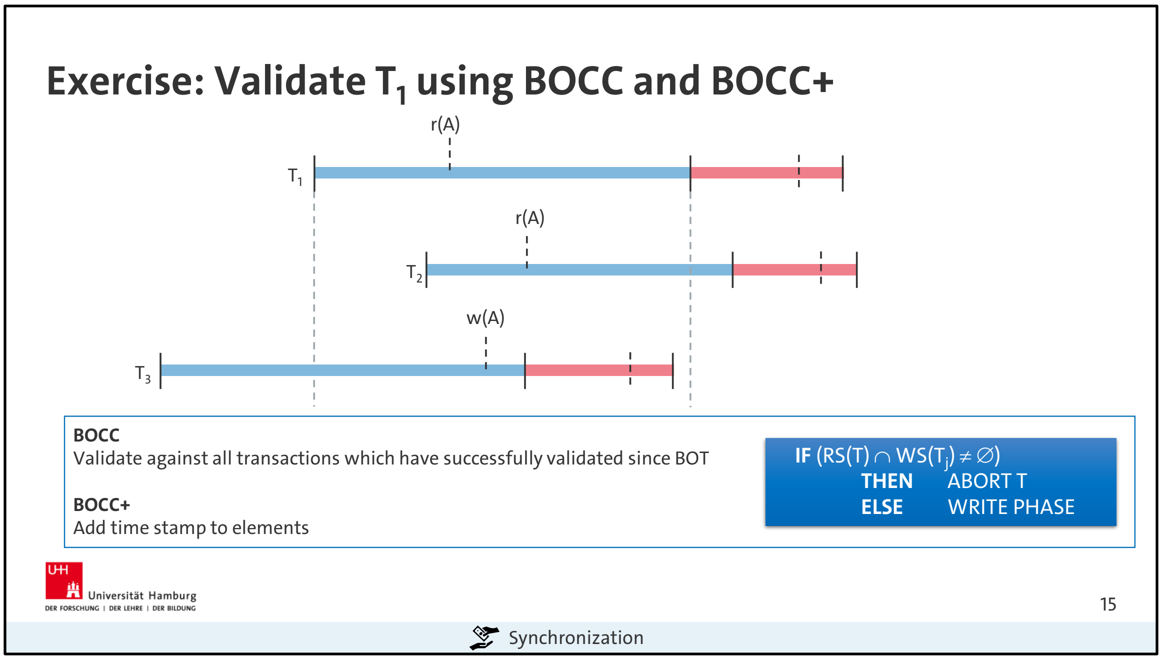
Synchronization

- T validates against T_j because T_j was running during the read phase of T
- T does not validate against T_k because T_k starts after the read phase of T has finished

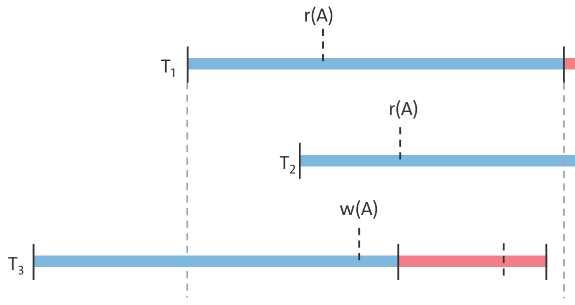
Advantages
- Early rollback possible → Less unnecessary work than in BOCC
- Choice which transaction to abort:
    - Kill approach: the transactions which are still running are aborted
    - Die approach: The validating transaction “dies” (is aborted)
- No need for keeping track of write sets, less work in validation phase

!Track number of rollbacks for each transaction to avoid that individual transactions are rolled back forever → High risk for long running transactions

Disadvantages
- Still a high number of rollbacks possible
- During validation and write phase, WS(T) must be “locked”, such that $RS(T_j)$ does not change



Exercise: Validate T_1 using BOCC and BOCC+



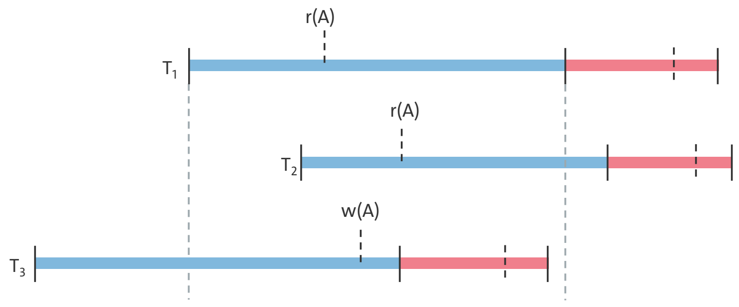
BOCC
Validate against all transactions which have successfully validated since BOT

BOCC+
Add time stamp to elements

```
IF (RS(T) ∩ WS(T_j) ≠ ∅)
    THEN    ABORT T
    ELSE    WRITE PHASE
```

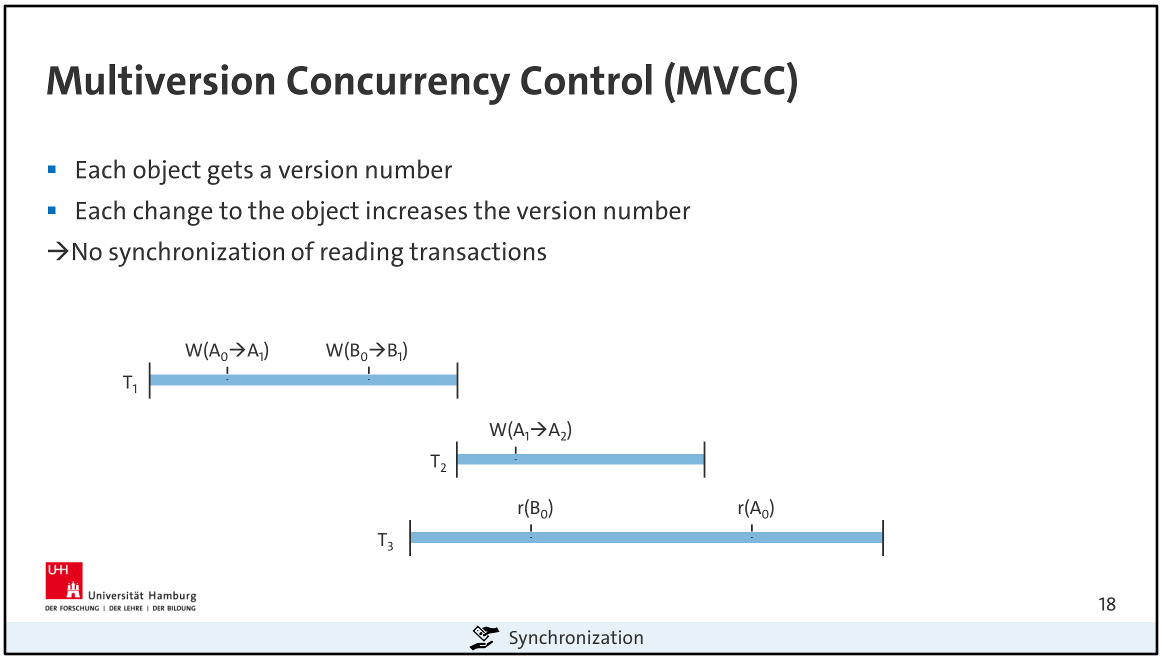
Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Exercise: Validate T_3 using FOCC



FOCC
Only writing transactions validate against running transactions

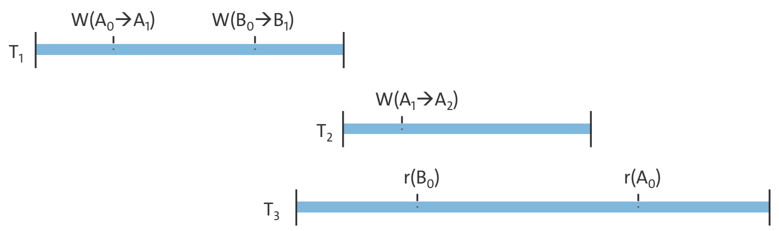
```
IF (WS(T) ∩ RS(T_j) ≠ ∅)
    THEN    ABORT T OR T_j
    ELSE    WRITE PHASE
```



Multiversion Concurrency Control (MVCC)

- Each object gets a version number
- Each change to the object increases the version number

$\rightarrow$No synchronization of reading transactions



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

18

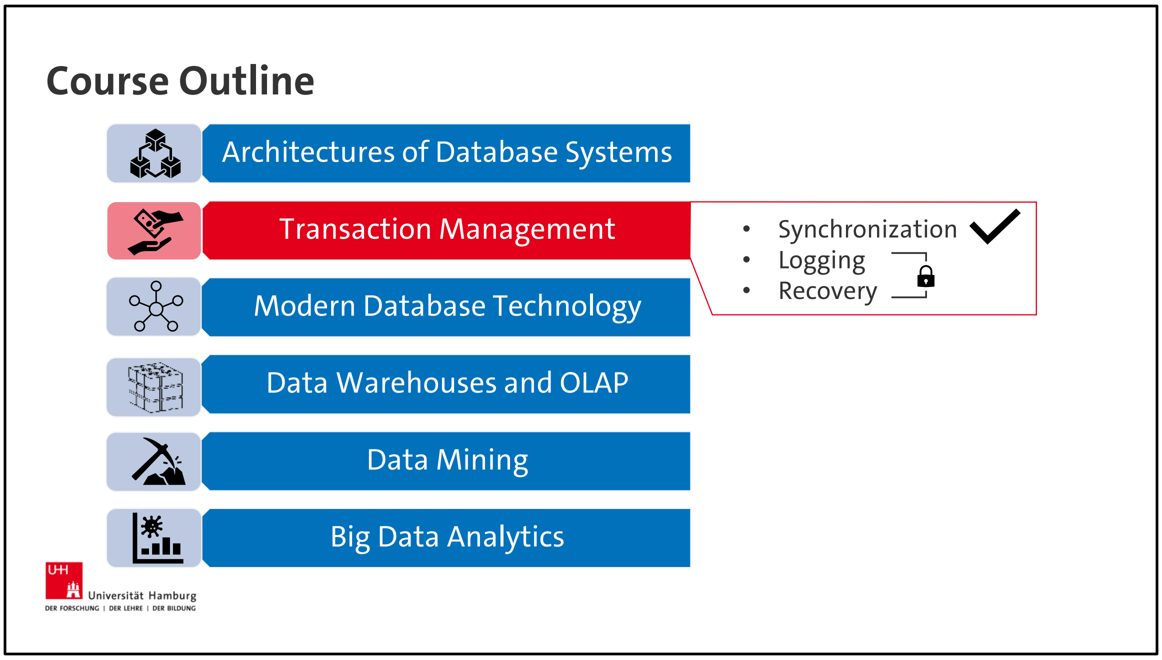
Synchronization

- Widely used approach, e.g. in PostgreSQL, Sap Hana, Oracle
- Risk of reading older versions of objects but smaller risk of discarded work done $\rightarrow$ no rollback
- Needs additional storage & maintenance, e.g. for version pool management, garbage collection
- Different methods for realization, e.g.:
    - Transactions read the version that is valid at their BOT/older than the time stamp of their BOT
    - Versions of an object are linked to easily find the most recent version before BOT
    - Version can be removed from the pool as soon as there is a newer version with a time stamp that is smaller than the oldest BOT of a reading transaction
    - Example on slide: T3 reads older versions because these were valid at its BOT, i.e. the newer versions were not committed, yet

!There are more methods for concurrency control, e.g. predicate locks (address the phantom problem), Locking on B*-trees, Escrow method, MVCC with snapshot isolation,...

→Further reading for the interested audience

- Cahill, Michael J., Uwe Röhm, and Alan D. Fekete. "Serializable isolation for snapshot databases." *ACM Transactions on Database Systems (TODS)* 34.4 (2009): 1-42.
- O'Neil, Patrick E. "The escrow transactional method." *ACM Transactions on Database Systems (TODS)* 11.4 (1986): 405-430.
- Eswaran, Kapali P., et al. "The notions of consistency and predicate locks in a database system." *Communications of the ACM* 19.11 (1976): 624-633. https://dl.acm.org/doi/pdf/10.1145/360363.360369



Course Outline

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

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG