# Transactions and Concurrency control

Transactions

Concepts

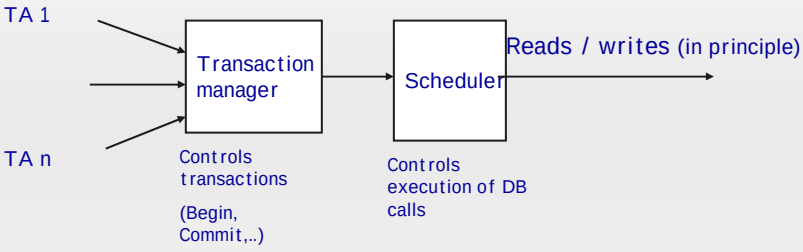
Serializability

Concurrency control

Locking
Multiversion cc
Optimistic cc

## Concurrency control ...and serializability

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




- Concurrency control in DBS: methods which schedule the operations of different TAs in a way which guarantees serializability of all transactions ("between system start and shutdown")

## Concurrency control — Principle methods

- Three principle concurrency control methods
  - Locking (most important)
  - Optimistic concurrency control
  - Time stamps
- 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
- Not considered here: transaction independent locking, e.g. writing a page p to disk requires a short term lock on p



## Concurrency control — Principle methods

- Locking is pessimistic:
  - Assumption: during operation op [x] of TA1 a (potentially) conflicting operation op'[x] of TA2 will access the same object x
  - This has to be avoided by locking x before accessing x

S: r1[y], r3[u], r2[y], w1[y], w2[x], w1[x], w2[z], c2, w3[x]



TA2 read x, wait until c2 has committed

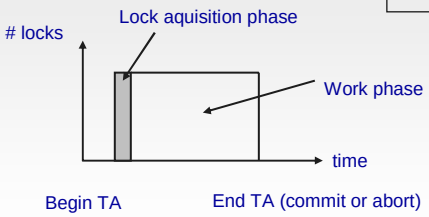
- An optimistic strategy would be:
  - Perform all operations on a copy of the data. Check at the end – before commit – if there were any conflicts.
  - If no: commit, else abort (rollback) - more or less

## Concurrency Control — Locking

- Lock protocols
  - Simplistic: Lock each object before writing, unlock afterwards ⇒ schedule will not be serializable (why?)
  - Preclaiming
    - Acquire all locks you need before performing an operation,
    - release, if you do not get all of them and try again race condition!
    - Execute transaction
    - Release locks

Preclaiming serializable?

Why (not) ?




Bad: objects to be processed may not be known in advance.

Not used in DBS.



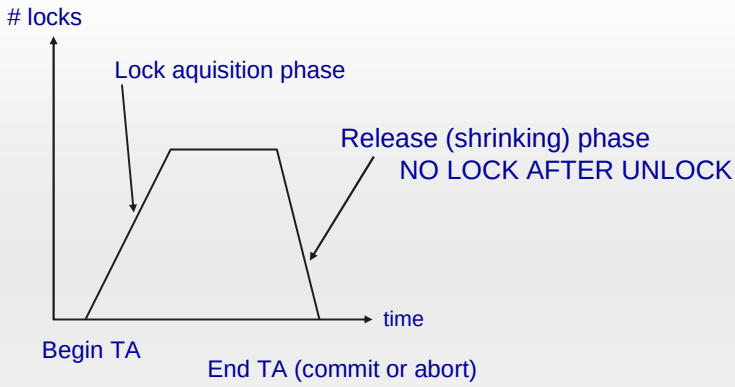
## Concurrency control — 2PL

Important method

- Two phase locking (2PL)
  1. Each object referenced by TAi has to be locked before usage
  2. Existing locks of other TA's will to be respected
  3. No lock is requested by a TAi , if a lock has been released by the same transaction TAi
     (no lock after unlock)
  4. Locks are released at least at commit time
  5. A requests of a lock by a TA which it already holds, has no effect.

     ... but a read lock may be "upgraded" to a write lock

## Concurrency control — 2PL




- Locked objects may be read / written already in lock aquisition phase

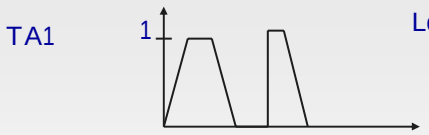


## Concurrency control — 2PL

- Why no lock after unlock?
- Example:

lock1(x), r1[x], x=x*10, ulock1(x)
lock2(x), r1[x], x:=x+1, w2[x] ulock2(x)
lock1(x), w1(x), ulock1(x)

results in a lost update




Lock profil of TA1 is NOT 2PL

⇒ Rule 3 is essential

- 2-Phase locking theorem

If all transactions follow the 2-phase locking protocol, the resulting schedule is serializable

- Proof sketch:
  - Suppose a resulting schedule is not serializable. when using 2PL ⇒ conflict graph contains a cycle ⇒ there are transactions TA1 and TA2 with conflict pairs (p,q) and (q', p'),
    p, p' atomic operations of TA1, q,q' of TA2, p,q access the same object x, and q', p' an object y (assuming a cycle of length 1, induction for the general case)

→





- Proof (cont.)
- Let e.g. (p,q) = (r1[x], w2[x]),
  (q', p') = (w2[y], w1[y])

Analyze all of the possible sequences of Execution:

p, q, q', p
p, q', q, p'
q', p, q, p
q', p, p', q
q', p', p, q

T2: Lock y, T1: Lock x, T2: Lock x, T1: Lock y

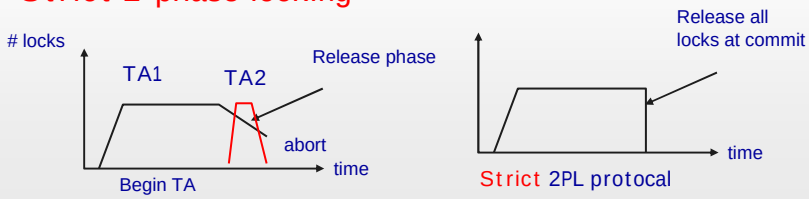
T1 must have released lock on x and aquired one on y (or T2 must have aquired after release)
Hurts 2-phase rule! Contradiction to assumption that all follow 2PL protocol

Same holds for the other possible sequences ⇒ Theorem

- Note: serializability does not imply 2PL, i.e. there are serializable schedules which cannot result from a scheduler employing 2PL

## Concurrency control — 2PL

- Strict 2-phase locking




- Another transaction TA2 could have used an object x which was unlocked by TA1 in the release phase
  - not a problem, if TA1 commits
  - if TA1 aborts, TA2 has used a wrong state of x
    TA2 has to be aborted by the system
- May happen recursively: cascading abort, bad ...
- Strict 2PL: Release all locks at commit point.



## Concurrency control — Deadlock

- Lock conflict
  - Two or more processes request an exclusive lock for the same object
- Deadlock
  - Locking: threat of deadlock
    - 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

## Concurrency control — Secondary goals

- 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, optimistic cc



## Concurrency control — Lock modes

- Lock modes and lock compatibility
  - RX – model: read (R) and write(X) locks

holder

| requester | R | X |
|---|---|---|
| R | + | - |
| X | - | - |

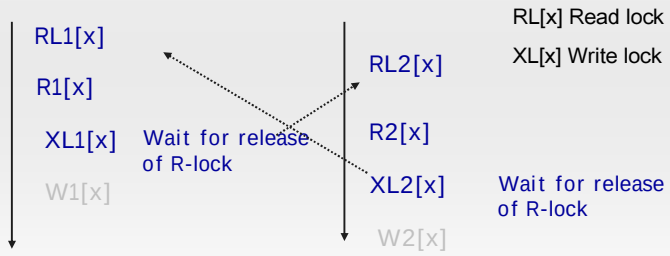
Lock compatibility matrix

- Lock compatibility in the RX model:
  - Objects locked in R-mode may be locked in R- mode by other transactions(+)
  - Objects locked in eXclusively mode (X-mode) may not be locked by any other transaction in any mode.

    Requesting TA must wait.

## Concurrency control — Reduce deadlock threat

- Deadlocks caused by typical read / write sequences
- TA1: read account_record x; incr(x.balance); write account_record
- TA2: read account_record x; incr(x.balance); write account_record




- Gives rise to Read-Update-Exclusive Model (RUX)



## Concurrency control — Lock modes

- RUX Lock protocol
  - Transactions which read and subsequently update an object y request a U-lock
  - U-locks are incompatible with U-locks ⇒ deadlock-thread avoided
  - U / R-lock compatibility asymmetric, why?

| requester \ holder | R | U | X |
|---|---|---|---|
| R | + | - | - |
| U | + | - | - |
| X | - | - | - |

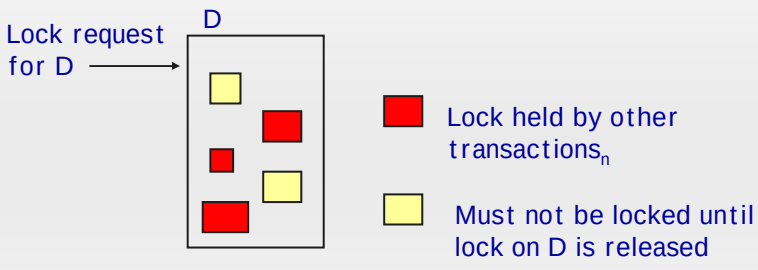
How does DBS know, that update is intended?

- Hierachical locking
  - One single lock granularity (e.g. records) insufficient overhead of many record locks large compared to a table lock
  - Most DBS have at least two lock granularities
  - Problem: $TA_i$ wants to lock table R
    - some rows of R are locked by different transactions
    - Different lock conflict than before: $TA_i$ is waiting for the release of all record locks
    - No other TA should be able to lock a record, otherwise $TA_i$ could starve



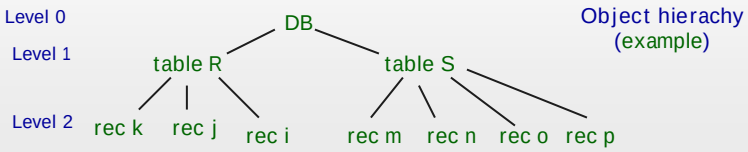


- Locks of different granularity




Efficient implementation of this type of situation??

- Intention locks

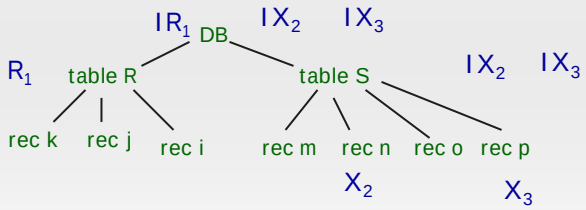



- Important feature of hierarchical locking: intention locks: for each lock mode, there is an intention lock, e.g. for RX-lock modes: IR and IX
- Semantics:
  A TA holds a IM-lock on an object D on level i, if and only if it holds an M-lock on an object D' on level j > i subordinate to D





- Hierachical locks :
  - 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

## Concurrency control — Lock modes

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

holder

| requester | IR | IX | R | X |
|---|---|---|---|---|
| IR | + | + | + | - |
| IX | + | + | - | - |
| R | + | - | + | - |
| X | - | - | - | - |

Compatibility matrix

How to combine with U-lock mode?



## Concurrency control — Deadlocks

- Deadlocks

  ... can happen with 2PL protocol (see above)
  - Release of a lock could break rule 4

    XL1[x] , XL2[y], XL1[y] -> TA1: WAIT for XU2[y] , XL2[x] -> TA2: WAIT for XU1[x]
  - Note: deadlock different from lock conflicts:

    .... XL1[x] , XL2[y], XL1[y] -> TA1: WAIT for XU2[y] XL2[z], w2[y], w2[z], XU2[y],...

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

Lock conflict resolved by XUnlock2[x], TA1 proceeds

Not schedules, but sequences of operations including lock / unlock, e.g. in a schedule

## Concurrency control — Deadlock

- Resolving deadlocks
  - Cycle check in Wait-for-graph
    - Waiting of TA1 for release of lock on x by TA2 is indicated by an arc from TA1 to TA2 labeled "x"
    - Cycles indicate deadlock
    - In a distributed environment, deadlocks may involve different systems. How to detect cycles?
    - One of the waiting transaction ("victim") is rolled back
    - Which one??
  - Timeout
    - If TA has been waiting longer than the time limit, it is aborted.
    - Efficient but may roll back innocent victims (deadlock does not exist)

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



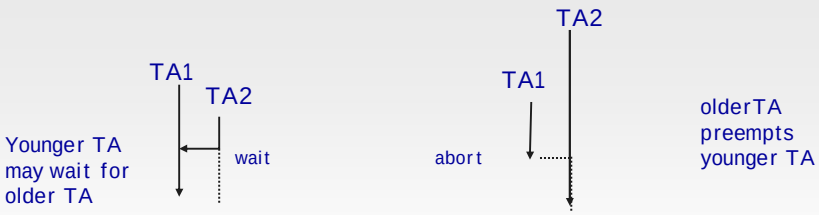
## Concurrency control — Deadlock avoidance

- Avoiding deadlocks
  - Deadlocks only occur, if no lock holder is preempted, i.e. looses his lock
  - Preemption may be forced by the lock manager
  - If TA t is preempted, it is forced to rollback
  - Preemption $\Rightarrow$ no deadlocks, but living transactions may be killed
- Wait/Die - Wound/Wait : Basic idea
  - Solve lock conflicts by rollback of one of the conflicting transactions....
  - .... but not always
  - Rollback dependent on the relative age of the transactions
  - Time stamp for each transaction

## Concurrency control — Deadlock avoidance

- Wound/Wait – Wait / Die methods
  - Each transaction has an initial timestamp TS(i)
  - If TA2 requests a lock on x and there is a lock conflict with TA1, then
  - WOUND / WAIT

    if ts(TA1) < ts(TA2) then TA2.WAIT else TA1.ABORT



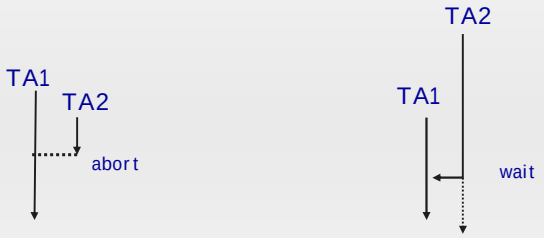

**If TS(TA1) < TS(TA2) then TA2.wait else TA1.abort**



## Concurrency control — Deadlock avaidance

- WAIT / DIE

  if ts(TA1) < ts(TA2) then TA2.ABORT else TA2.WAIT

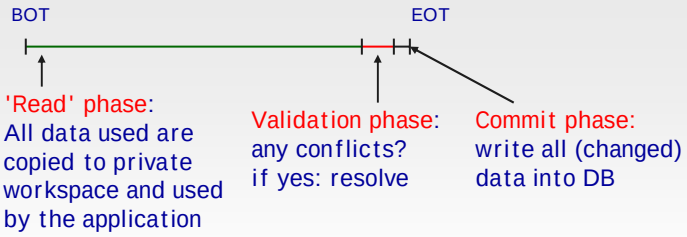



**If TS(TA1) < TS(TA2) then TA2.abort else TA2.wait**

No deadlocks! Why?

Aborted transaction restarts with old timestamp in order to avoid starvation

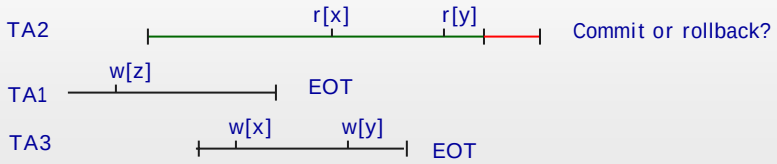
- Optimistic concurrency control
  - Locks are expensive
  - If conflicts are rare, retrospective check for conflicts and – if any – abort are cheaper, hopefully
  - Basis idea: let all transactions work on copies, at the end, check for conflicts, commit (or abort)







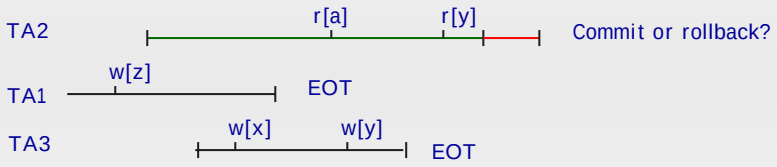

- Backward oriented concurreny control (BOCC)




- ReadSet R(T) = set of data, transaction T read in read phase
- WriteSet W (T) = set of data, T has changed during read phase
  Assumption: $W(T) \subseteq R(T)$
  Example above: $x,y \in R(T2)$, $x,y \in W(T3)$, $z \in W(T1)$

- BOCC_validate(T) :

  if for all transactions (T') which committed after BOT(T) :
  R(T) ∩ W(T') = ∅
  then T.commit // successful validation
  else T.abort




Note :
Conflict caused by r[y] is NOT harmful,
but rollback of TA2 because R(TA2) ∩ W(TA3) != ∅

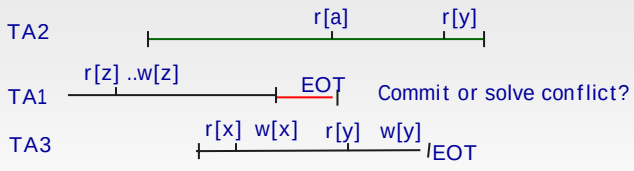
=> Unnecessary Aborts



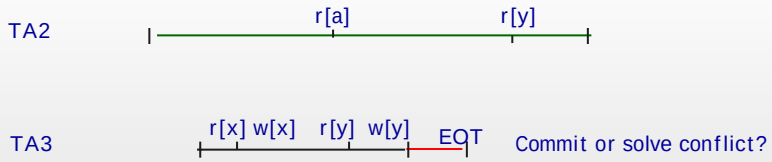


- Forward oriented optimistic Concurrency control (FOCC)
  - Forward looking validation phase:

    if there is a running transaction T' which read data written by the validating transaction T then solve the conflict (e.g. kill T'), else commit



## Concurrency control — Optimistic CC



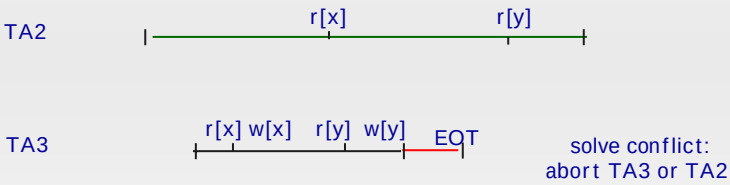

FOCC_validate(T) : if for all running transactions (T')
$R^n(T') \cap W(T) = \varnothing$
then T.commit // successful validation
else solve_conflict ( T, T')

$R^n(T')$: Read set of T' at validation time of T (current read set)



## Concurrency control — Optimistic CC

- Validation of read only transactions T:
  FOCC guarantees successful validation !
- FOCC has greater flexibility
  Validating TA may decide on victims!




- Issues for both approaches:
  fast validation – only one TA can validate at a time.
  Fast and atomic commit processing,
- Useful in situation with few expected conflicts

- Combining locks with optimism
  - Example: high traffic reservation system
  - typical TA: check "seats_avail >0 ?" //seats_avail is hot spot
    if yes, do this and that;
    write seats_avail-1
  - seats_avail is a hot spot object
  - Not the state per se is important but the predicate "seats_avail >0 ?"
  - Optimism: if pred is true at BOT then it will be true with high probability at EOT
  - But if not: abort





- Additional operations Verify and Modify:

```
Verify P : check predicate  P  ("seats_avail >0"?)
           //like read phase
 put "seats_avail-1" into to_do list
 rest of TA
 EOT:
 Modify : for all  operations on to_do list
          { lock; verify once more;
            if 'false' rollback else write updates;}
           unlock all;
```

Roughly the approach taken in IMS / Fast Path

- Short locks, more parallelism
- If only decrement / increment operations: concurrent writing possible without producing inconsistencies

## Concurrency control — Multiversion concurrency

- Multiversion CC:


Arrows from TA2-ops to conflicting TA1-ops

not serializable.

If r1[y] had arrived at the scheduler before w2[y] the schedule would have been serializable.

- Main idea of multiversion concurrency control : Reads should see a consistent (and committed) state, which might be older than the current object state.
- Necessary: Different version of an object
- Read and write locks compatible (!)
- Example: TA2 must not write its version of y before TA1 has release lock on y



## Concurrency control — Multiversion concurrency

- Lock based MVCC
  - Read locks always granted, write lock if object not write locked
    =>
    two versions: consistent one and writable private copy
  - When TA wants to write modified copy of x into DB it has to wait until all reader of x have released read lock
  - write is delayed to ensure consistent read using a certify lock

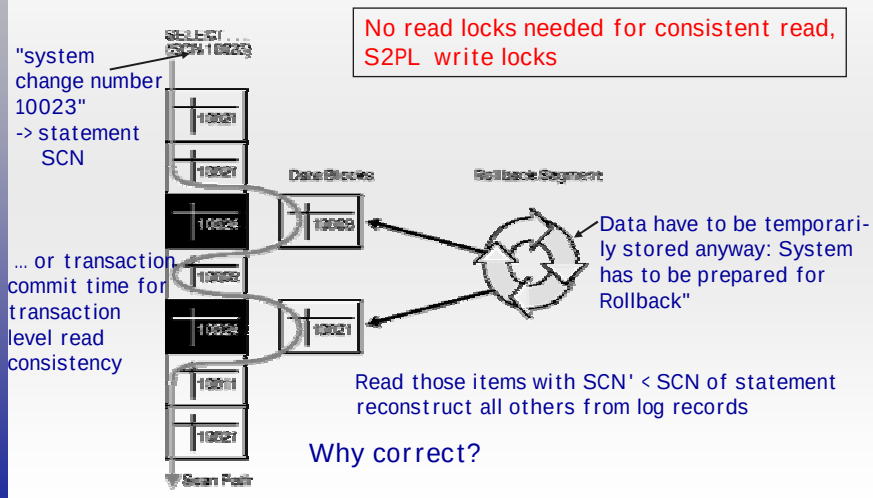
| | R | W | C |
|---|---|---|---|
| R | + | + | - |
| W | + | - | - |
| C | - | - | - |

C = Certify

Read locks needed ??

## Concurrency control — Multiversion concurrency

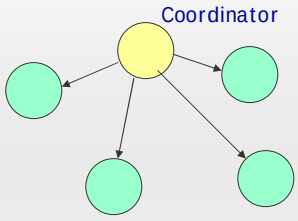
- Read Only Multiple version CC (used in Oracle)






## Distributed Transactions — Intermezzo

- Configuration



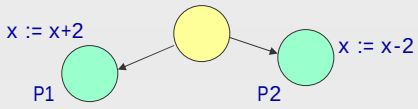

- Different resource managers involved in the transaction
  e.g: database systems, mail server, file system, message queues,...
- Asynchronous and independent
- One transaction coordinator can be a resource manager or not
- One or more participants

- Examples: Transfer of money/shares / ... from Bank A to B
  ECommerce systems
  All kinds of processing in decentralized organizations

- Frequently used in multi-tier architectures: middle tier accesses different databases
  Just open two or more JDBC-connections

## Distributed Transactions — Intermezzo

- Problems
  - No problem ... if all systems work reliable
  - Deadlock? Difficult to detect: use optimistic locking
  - Obvious inconsistencies, if one participant commits, another crashes:



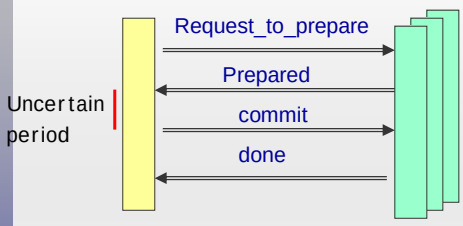

Coordinator: COMMIT
P1 commits
P2 crashes, undo??
Introduces global inconsistency

- Assumptions
  - Each resource manager has a transactional recovery system (log operations, commit, rollback)
  - There is exactly one commit coordinator, which issues commit for a transaction exactly once
  - A transaction has stopped processing at each site before commit is issued



## Distributed Transactions

- The Two Phase Commit protocol (2PC)

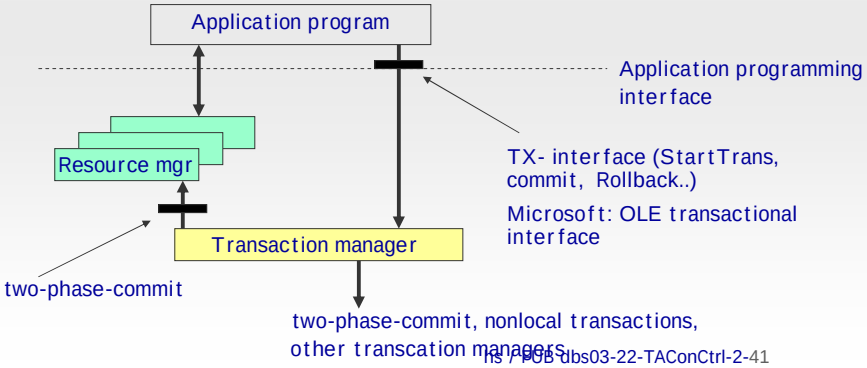



1. Coordinator asks for preparing commit of a transaction
2. If (all participants answer 'prepared')
3. Coordinator asks for commit
   else asks for abort
4. Participants send 'done'

- After prepare phase: participants are ready to commit or to abort; they still hold locks
- If one of the participants does not reply or is not able to commit for some reason, the global transaction has to be aborted.
- Problem: if coordinator is unavailable after the prepare phase, resources may be locked for a long time

## Distributed Transactions

- Transaction managers: the X/Open transaction model
  - Independent systems which coordinate transactions involving multiple resource managers as a service for application programs






## Summary: Transactions and concurrency

- Transactions: Very import concept
- Model for consistent, isolated execution of TAs
- Scheduler has to decide on interleaving of operations
- Serializability: correctness criterion
- Implementation of serializability:
  - 2-phase-locking
  - hierarchical locks
  - variation in order to avoid deadlocks : wound / wait, wait / die
  - optimistic concurrency control
  - multiversion cc