On Building Integrated and Distributed Database Systems

Transaction processing in DDBS

Robert Wrembel
Poznań University of Technology
Institute of Computing Science
Poznań, Poland
Robert.Wrembel@cs.put.poznan.pl
www.cs.put.poznan.pl/rwrembel

Transaction

- A unit of interaction between a user and a database
  - consists of set of commands
- States
  - committed
  - aborted/rolled back
- Properties (ACID)
  - atomicity
  - consistency
  - isolation
  - durability

Update anomalies (1)

- Concurrent access of multiple users to the same data
  - reading
  - writing
- Dirty read

transaction T1	transaction T2
read(account_1) account_1=100	
account_1:=account_1+20 account_1=120	
write(account_1)	
	read(account_1) account_1=120
	account_1:=account_1+40 account_1=160
rollback	

Update anomalies (2)

- Lost update

transaction T1	transaction T2
read(account_1) account_1=100	
	read(account_1) account_1=100
account_1:=account_1+10 account_1=110	
	account_1:=account_1+40 account_1=140
write(account_1)	
	write(account_1)

Update anomalies (3)

- **Nonrepetable reads**
  - T1 reading the same row multiple times sees its changes

transaction T1	transaction T2
read(account_1) account_1=100	
	account_1:=account_1+40 account_1=140
read(account_1) account_1=140	
	account_1:=account_1+20 account_1=160
read(account_1) account_1=160	
	commit

Update anomalies (4)

- **Inconsistent reads (inconsistent analysis)**

transaction T1	transaction T2
read(account_1) account_1=100	
account_1:=account_1-20 account_1=80	
write(account_1)	
read(account_2) account_2=200	
	read(account_1+account_2) account_1+account_2=280
account_2:=account_2+20 account_2=220	
write(account_2)	
read(account_1+account_2) account_1+account_2=300	read(account_1+account_2) account_1+account_2=300

Update anomalies (5)

- **Phantoms**
  - if one executes a query at time T1 and re-execute it at time T2, additional rows may have been added to the database, which may affect the results;
  - it differs from a nonrepeatable read in that with a phantom read, data you already read hasn't been changed, but instead, more data satisfies your query criteria than before

transaction T1	transaction T2
read(table Accounts where balance>9000) account_1 account_2	
	insert account_3 (balance=10000)
	commit
read(table Accounts where balance>9000) account_1 account_2 account_3	

Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
READ UNCOMMITTED	Permitted	Permitted	Permitted
READ COMMITTED	--	Permitted	Permitted
REPEATABLE READ	--	--	Permitted
SERIALIZABLE	--	--	--

- **Serializable**
  - it prevents other users from updating or inserting rows into the data set until the transaction is complete

Transaction management

- Scheduling ⇨ preserving consistency
- Deadlock detection
- Commit protocol

Transaction scheduling

- **Schedule**
  - a sequence of operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions
  - let schedule S consists of a sequence of the operations from the set of T1, T2, ... Tn transactions
  - for each transaction Ti in schedule S the original order of operations in Ti must be the same in S
- **Serial schedule**
  - a schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions
- **Nonserial schedule**
  - a schedule where the operations from a set of concurrent transactions are interleaved

Serializability

- Anomalies 1-5 resulted from parallel execution of transactions
- Serial execution prevents from the anomalies
- Objective - serializability:
  - find nonserial schedules that allow transactions to execute concurrently in a way producing a database state the same as produced by a serial execution
- If a set of transactions executes concurrently, we say that a nonserial schedule is correct if it produces the same results as a serial schedule ⇨ such a schedule is called serializable

Concurrency control techniques

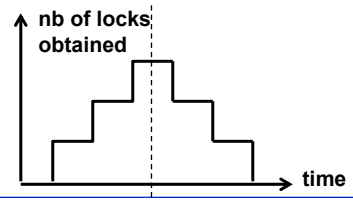
- Methods used for ensuring serializability of concurrent transactions
  - Locking
  - Timestamping
  - Optimistic method

Locking

- A transaction has to obtain a shared (read) lock or an exclusive (write) lock on a data item before executing an operation on that item
- Two-phase locking protocol (2PL)
  - growing phase
    - all required locks are obtained
    - no obtained locks are released
  - shrinking phase
    - all obtained locks are released
    - no new locks are obtained

lock compatibility

	shared	exclusive
shared	✓	x
exclusive	x	x



Multiversion locking

- Versions of data are maintained by a system
  - record R being updated has its version R_{old} before update
  - transactions reading R (that is modified) access R_{old}

lock compatibility

	IS	IX
IS	✓	✓
IX	✓	x

- Locking methods
  - prevent conflicts by making transactions wait
  - may cause in deadlocks

Timestamping (1)

- No waiting transactions
- Conflicting transactions are rolled back and restarted
- Timestamp
  - transaction timestamp: a unique identifier created by a dbms that indicates the relative starting time of a transaction
  - data timestamp
    - read_timestamp - TS of the last transaction that read a record
    - write_timestamp - TS of the last transaction that modified a record

Timestamping (2)

- Timestamping
  - a concurrency control protocol that orders transactions in such a way that older transactions (with smaller timestamps) get priority in the event of conflict
- Transaction T is allowed to read or write record R if the last update of R was done by an older transaction
  - otherwise T is restarted and assigned a new TS
  - new (younger) TS prevents T from continuously being restarted

TS ordering protocol (1)

- Transaction T issues READ(x)
  - x has already been updated by a younger transaction: TS(T) < write_timestamp(x)
    - rollback T
    - assign a new TS to T
    - restart T
  - TS(T) >= write_timestamp(x)
    - READ(x)
    - assign read_timestamp(x):=max{TS(T), read_timestamp(x)}

TS ordering protocol (2)

- Transaction T issues WRITE(x)
  - x has already been read by a younger transaction: TS(T) < read_timestamp(x)
    - rollback T
    - assign a new TS to T
    - restart T
  - x has already been written by a younger transaction: TS(T) < write_timestamp(x)
    - rollback T
    - assign a new TS to T
    - restart T
  - otherwise: execute T
    - assign write_timestamp(x) = TS(T)

Optimistic protocol (1)

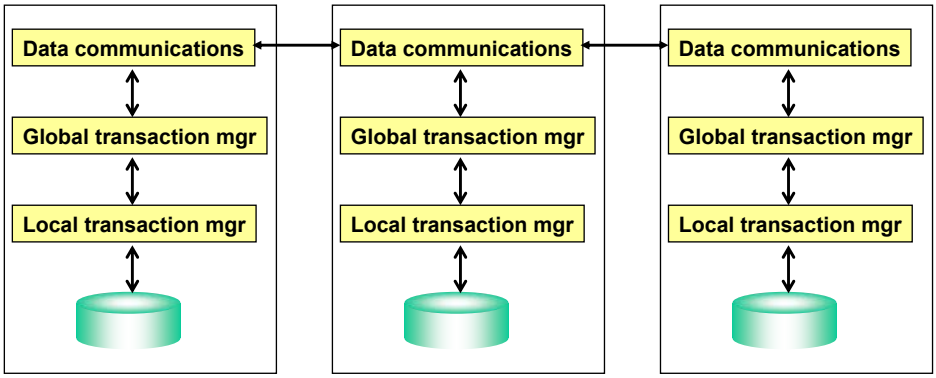
- Assumptions: conflicts are rare
  - when transaction T is going to commit, a check is performed to determine whether a conflict occured
  - if so T has to be aborted and restarted
- Offers higher concurrency as no locking is used
- Costs of restarting conflicting transactions
- Phases
  - read: reading and updating data
    - local copies of updated data are used
    - transaction T gets assigned the START(T) timestamp
  - validation
    - checking if serializability is not violated
    - transaction T gets assigned the VALIDATION(T) timestamp
  - write: writing updated data on disk
    - transaction T gets assigned the FINISH(T) timestamp

Optimistic protocol (2)

- To pass the validation phase the following has to be true

1. All transactons T_i with earlier timestamps must have finished before transaction T started: FINISH(T_i) < START(T)
2. If T starts before earlier transaction T_e finishes, then
   a) the set of records written by T_e is not the same as read by T and
   b) T_e completes its write phase before T enters its validation phase: START(T)<FINISH(T_e)<VALIDATION(T)
      - it guarantees that writes are done serially, ensuring no conflicts

Distributed transaction management



Distributed concurrency control

- **Distributed serializability**
  - **if the schedule of transactions at each site is serializable then the global schedule (the union of local schedules) is also serializable if:**
    - $T_m^{S1} < T_n^{S1}, T_m^{S2} < T_n^{S2}, ..., T_m^{Sn} < T_n^{Sn}$
      - m, n – is the number of a transaction T
      - S_i – is a site
- **Methods**
  - **locking protocols**
    - **centralized 2PL**
    - **primary copy 2PL**
    - **distributed 2PL**
    - **majority locking**
  - **timestamping**

Centralized 2PL (1)

- A single site maintains all locking info
- Only one lock manager in a distributed db can set and release locks
- 2PL for a global transaction initiated at site S1 - Steps
  1. Transaction coordinator at S1 divides transaction T into a number of subtransactions
     - If T updates a replica, then the coordinator must ensure that all copies are updated - exclusive locks requested on replicas
     - Elect a replica for read (usually the local one)
  2. Local transaction manager sets and releases locks as instructed by the centralized lock manager (normal 2PL is used at local sites)
  3. Centralized lock manager controls lock compatibility; incompatible requests are put into a queue

Centralized 2PL (1)

- Straightforward implementation
- Easy deadlock detection - one lock manager maintains all locks
- Low communication costs
- All lock info go to a single site - bottleneck
- Lower reliability - one central site

Primary copy 2PL

- For replicated data one copy is chosen as a primary one
- Distributing lock managers to several sites (not all)
- Update is executed first on the primary copy
  - trans. coordinator must determine the location of the primary copy and send locking request to its lock mgr
  - only the primary copy is exclusively locked
- The protocol guarantees that only the primary copy is current
  - once the primary copy is updated, the update is propagated to other copies, not necessarily as the same transaction
- May be used when
  - updates are infrequent
  - sites do not always need the latest data version
- More difficult to detect deadlocks

Distributed 2PL

- Distributing lock managers to every site
- A local lock manager is responsible for managing locks at its site
- For non replicated data Distributed 2PL is equivalent to standard 2PL
- For replicated data:
  - any replica can be used for reading
  - all replicas must be exclusively locked before updating
- Detecting and handling deadlocks is complex
- No bottleneck of centralized 2PL

Majority locking

- Distributing lock managers to every site
- A local lock manager is responsible for managing locks at its site
- For replicated data:
  - before updating a replica, a transaction has to obtain locks on over than the half of the replicas (instead of locks on all replicas like in distributed 2PL)
  - update propagation to unlocked replicas may be performed later
- Lower numer of sites locked
- Detecting and handling deadlocks is complex

Timestamp protocol

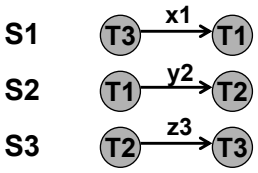
- Objective: to order transactions globally in such a way that older transactions (with smaller TS) get priority in the event of conflicts
- Unique TS generated locally and globally - problem
  - system clock - not synchronized
  - logical clock (incremented) - duplicate values at many sites
  - solution: local_TS || site_id
- Busy sites will generate younger TS ⇨ solution: TS synchronization between sites
  - each site includes its TS in inter-site messages
  - on receiving a message a site compares its TS (localTS) to the received TS (remoteTS)
    - if localTS<remoteTS set localTS:=remoteTS + increment_by
    - else do nothing

Distributed deadlock management

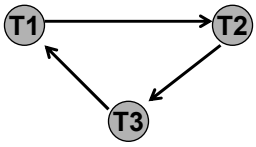
- Let us consider 3 transactions
  - T1 initiated at site S1 and requesting operation at S2
  - T2 inititated at S2 and requesting operation at S3
  - T3 inititated at S3 and requesting operation at S1
- Transaction locks

Time	S1	S2	S3
t1	read_lock(T1, x1)	write_lock(T2, y2)	read_lock(T3, z3)
t2	write_lock(T1, y1)	write_lock(T2, z2)	
t3	write_lock(T3, x1)	write_lock(T1, y2)	write_lock(T2, z3)

local WFG



global WFG



Deadlock detection

- In DDBS it is not sufficient for each site to build its local WFG
- It is necessary to construct a global WFG that is the union of local WFGs
- 3 methods for handling deadlock detection in DDBS
  - centralized
  - hierarchical
  - distributed

Exercise 1

- **Consider 5 transactions T1, T2, T3, T4, and T5**
  - **T1 initiated at site S1 and requesting operation at site S2**
  - **T2 initiated at site S3 and requesting operation at site S1**
  - **T3 initiated at site S1 and requesting operation at site S3**
  - **T4 initiated at site S2 and requesting operation at site S3**
  - **T5 initiated at site S3**

	S1	S2	S3
1.	write_lock(T3, x2)	read_lock(T4, x7)	
2.	read_lock(T1, x1)		write_lock(T4, x8)
3.	read_lock(T1, x2)		
4.	write_lock(T2, x1)	write_lock(T3, x7)	write_lock(T1, x8)
5.		write_lock(T5, x7)	
6.			write_lock(T5, x5)
7.			read_lock(T3, x5)

Exercise 1

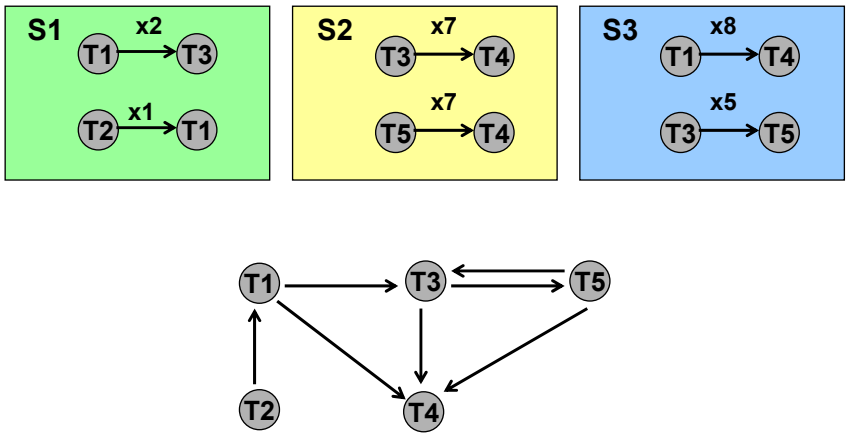
- **Locking information for these transactions are as follows**

Transaction	Data locked by transaction	Data transaction is waiting for	Site involved in operation
T1	x1	x8	S1
T1	x6	x2	S2
T2	x4	x1	S1
T2	x5		S3
T3	x2	x7	S1
T3		x3	S3
T4	x7		S2
T4	x8		S3
T5	x3	x7	S3

A. Create WFGs for each of the sites

B. Create a global WFG to check if there is a global deadlock

Solution



Centralized deadlock detection (1)

- A single site is appointed as the Deadlock Detection Coordinator (DDC)
- DDC is constructing and maintaining a global WFG
- Each lock manager periodically transmits its local WFG to the DDC
- DDC is checking for cycles in a global WFG
- If one or more cycles exist, then the DDC has to break each selected transaction to be rolled-back and restarted
- DDC has to inform all the sites involved in processing these transactions that they have to be rolled back and restarted

Centralized deadlock detection (2)

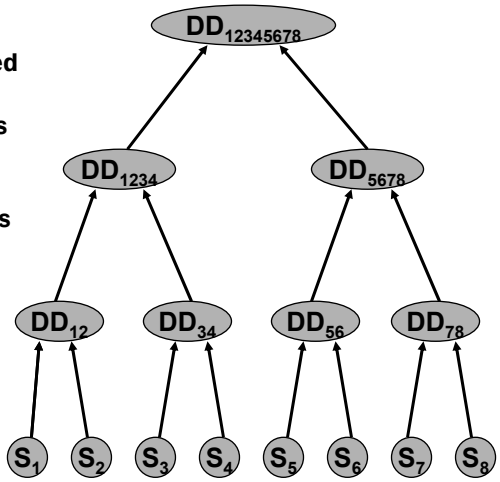
- To minimize the amount of data sent, a local lock manager sends only changes in WFG
- Such an architecture is less reliable as a failure of a central site makes deadlock detection impossible

Hierarchical deadlock detection (1)

- Sites are organized hierarchically
- Each site sends its local WFG to a deadlock detection site above in hierarchy
- It reduces the dependence on a centralized detection site, reducing communication costs
- More complex implementation

Hierarchical deadlock detection (2)

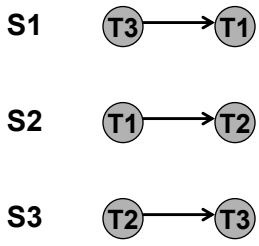
- At leaves (level 1) local deadlock detection is performed
- Level 2 nodes check for deadlocks in two adjacent sites (1-2 and 3-4)
- Level 3 nodes check for deadlocks in four adjacent sites (1-2-3-4 and 5-6-7-8)
- The root node is a global deadlock detector



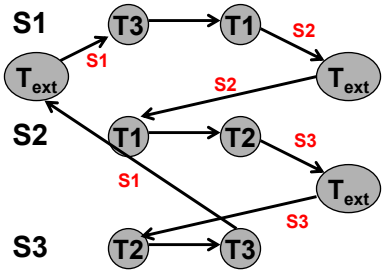
Distributed deadlock detection (1)

- External node T_{ext} is added to a local WFG to indicate an agent at a remote site
- When transaction T1 at site S1 creates an agent at site S2 then an edge is added to the local WFG from T1 to T_{ext}
- At site S2 and edge is added to te local WFG from T_{ext} to the agent of T1

local WFG

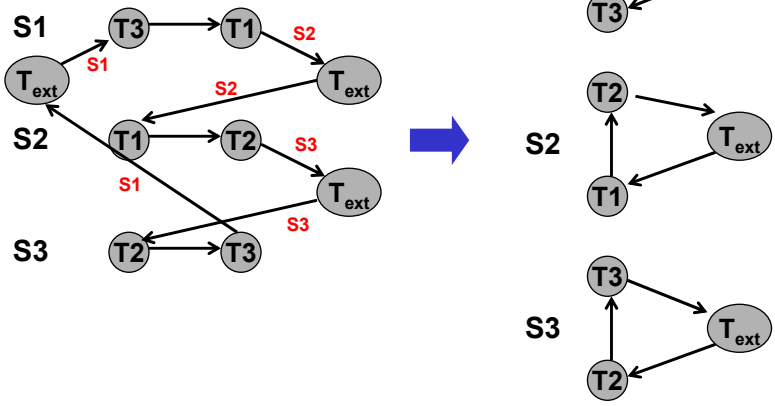


local WFG for distributed deadlock detection



Distributed deadlock detection (2)

local WFG for distributed deadlock detection

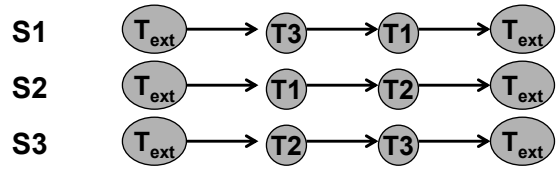


Distributed deadlock detection (3)

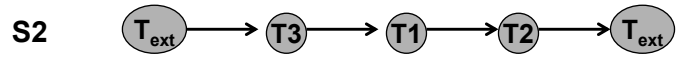
- If local WFG contains a cycle that does not include T_{ext} then the site has a deadlock ⇨ a DDBS has a deadlock
- A global deadlock potentially exists if the local WFG contains a cycle involving T_{ext}
  - since T_{ext} may represent different agents, the existence of such a cycle does not mean that there is a deadlock
  - to determine if there is a deadlock WFGs are being merged
- Transmitting a WFG
  - site S1 transmits its WFG only to the site (S_i) for which transaction T_k (at S1) is waiting
  - S_i receives the WFG from S_1 and merges it with its own WFG
  - after merging WFGs, S_i check if there is a cycle not involving T_{ext}
  - if a cycle is not found, then the merged WFG is transmitted to another site

Distributed deadlock detection (4)

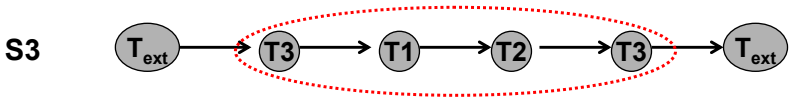
local WFG



➲ T1 is waiting for T2 ⇨ WFG is transmitted from S1 to S2 and merged at S2



➲ T2 is waiting for T3 ⇨ WFG is transmitted from S2 to S3 and merged at S3



cycle not involving Text ⇨ global deadlock

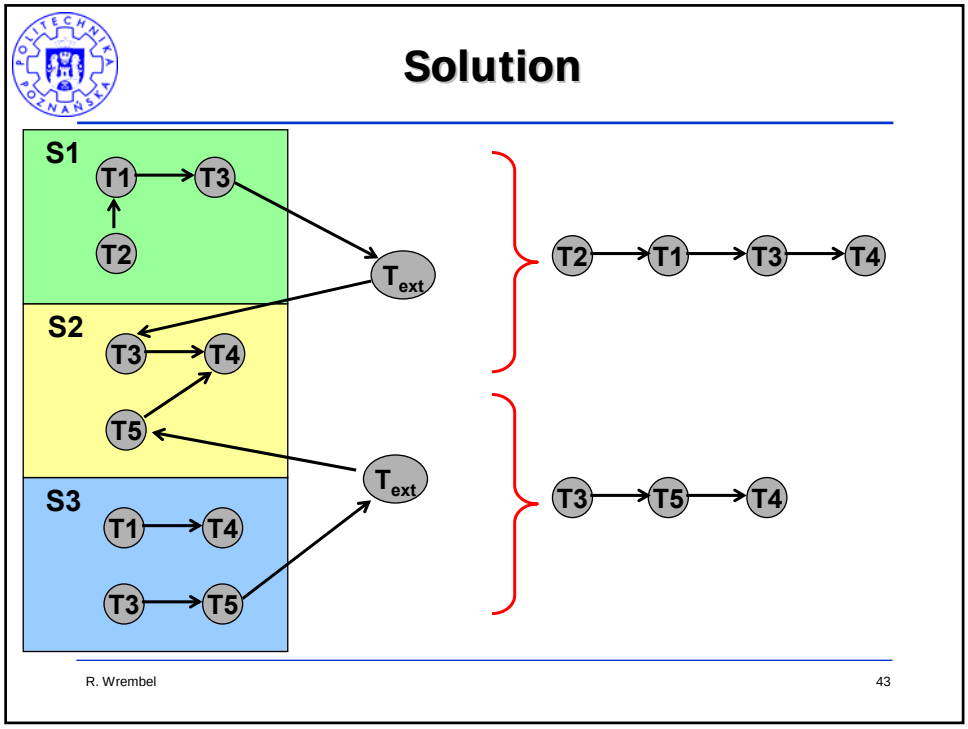
Exercise 2

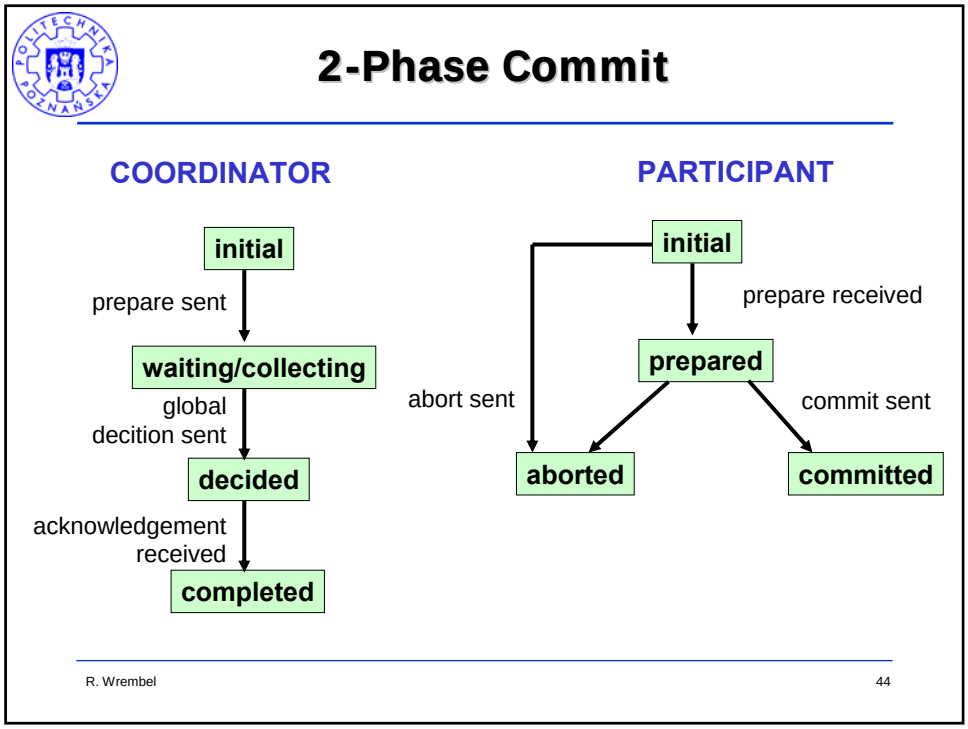
➲ Locking information for transactions are as follows

Transaction	Data locked by transaction	Data transaction is waiting for	Site involved in operation
T1	x1	x8	S1
T1	x6	x2	S2
T2	x4	x1	S1
T2	x5		S3
T3	x2	x7	S1
T3		x3	S3
T4	x7		S2
T4	x8		S3
T5	x3	x7	S3

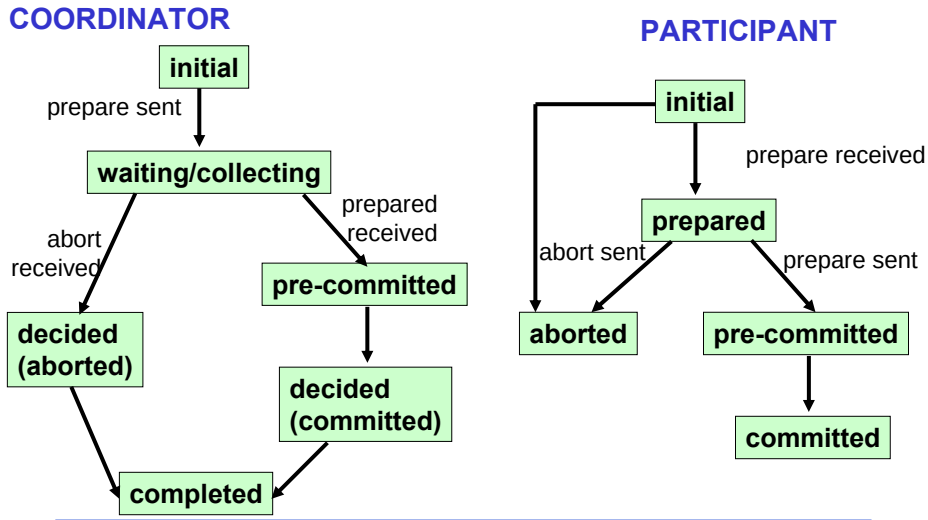
	S1	S2	S3
1.	write_lock(T3, x2)	read_lock(T4, x7)	
2.	read_lock(T1, x1)		write_lock(T4, x8)
3.	read_lock(T1, x2)		
4.	write_lock(T2, x1)	write_lock(T3, x7)	write_lock(T1, x8)
5.		write_lock(T5, x7)	
6.			write_lock(T5, x5)
7.			read_lock(T3, x5)

➲ Using the distributed detection algorithm check whether a deadlocks takes place


Solution
S1
T1
T3
T2
S2
T3
T4
T5
S3
T1
T4
T3
T5
T ext
T ext
T2
T1
T3
T4
T3
T5
T4
R. Wrembel
43


2-Phase Commit
COORDINATOR
PARTICIPANT
initial
prepare sent
waiting/collecting
global
decition sent
decided
acknowledgement
received
completed
initial
prepare received
prepared
abort sent
commit sent
aborted
committed
R. Wrembel
44

3-Phase Commit (1)



3-Phase Commit (2)

- Both the coordinator and participants still have periods of waiting, but the important feature is that all operational processes have been informed of a global decision to commit by the PRE-COMMIT message prior to the first process committing
- Participants can act independently in the event of failure