Concurrency Control

Concurrency Control

- Executing all transactions strictly serially (i.e., one after another) is safe, but slow
- Often system resources are not fully utilized, because a transaction waits for disk access or user input
- This transaction then blocks all other transactions
- To utilize system resources, concurrency is the natural choice

Concurrency Control(2)

- Unprotected concurrency, however, can lead to the following problems:
  - lost update
  - dirty read
  - non-repeatable read
  - phantom problem

Lost Update

T_1	T_2
bot	
$r_1(x)$	
$\hookrightarrow$	bot
	$r_2(x)$
$w_1(x)$	$\hookleftarrow$
$\hookrightarrow$	$w_2(x)$
commit	$\hookleftarrow$
$\hookrightarrow$	commit

The result of transaction T_1 has been lost!

Dirty Read

T_1	T_2
bot	
$\hookrightarrow$	bot
	$r_2(x)$
	$w_2(x)$
$r_1(x)$	$\hookleftarrow$
$w_1(y)$	
commit	
$\hookrightarrow$	abort

T_1 reads a value for x that is not valid!

Non-Repeatable Read

T_1	T_2
bot	
$r_1(x)$	
$\hookrightarrow$	bot
	$w_2(x)$
	commit
$r_1(x)$	$\hookleftarrow$
...	

T_1 reads x twice with different results!

Phantom Problem

T_1	T_2
bot	
`select count(*)`	
`from R;`	
↪	bot
	`insert into R …;`
	commit
`select count(*)`	↩
`from R;`	
…	

T_1 finds an additional tuple when processing the second query!

Avoiding the Problems

- Ideally, all of these problems should be avoided
- However, one has to make a trade-off between performance and accuracy
- The more safety, the slower the execution
- Using *isolation levels* one can tell the DBMS which level of safety is desired

- Setting the properties of a transaction:

set transaction *level, access mode*

- The following isolation levels are possible:
  - read uncommitted
  - read committed
  - repeatable read
  - serializable
- Possible access modes:
  - read only
  - read write

	lost update	dirty read	nonrep. read	phant. probl.
read uncommit.	(√)			
read committed	√	√		
repeat. read	√	√	√	
serializable	√	√	√	√

Transactions and SQL(3)

- “read only” tells the DBMS that a transaction contains only read operations
- This has an impact on performance
- Concurrent execution of read-only transactions is uncritical, i.e., any number of such transactions can run in parallel without any restrictions
- Only when a transaction that also writes joins in do precautions have to be taken

Transactions and SQL(4)

- Command to mark the beginning of a transaction

start transaction;

- Command for successful termination

commit [work];

- Command to abort

rollback [work];

What Does a DBMS Do?

- To better understand the solution approaches, we first look at the problem somewhat more formally
- Afterwards, we present solutions that are used in DBMSs

Formal Definition of a Transaction

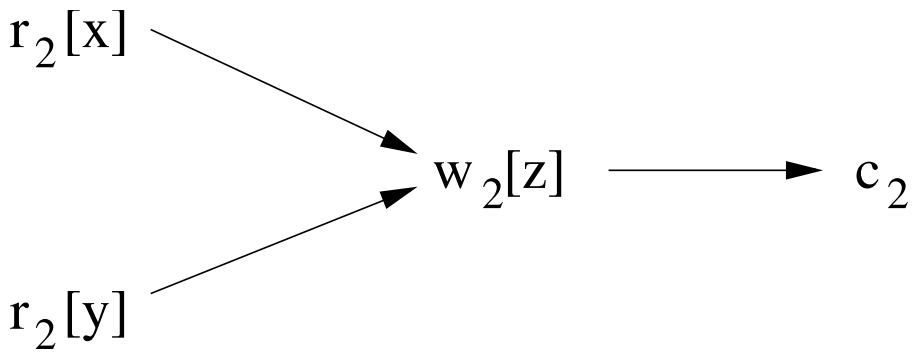
- Operations of a transaction T_i
  - $r_i(A)$: reading data item A
  - $w_i(A)$: writing data item A
  - a_i: abort
  - c_i: successful termination (commit)

  - *bot*: begin of transaction (implicit)

Formal Definition of a Transaction(2)

- A transaction T_i is a partial order of operations with the ordering relation $\underset{i}{<}$, such that:
  - $T_i \subseteq \{r_i[x], w_i[x] \mid x \overset{\text{is a data item}}{} \}$
    $\cup \{a_i, c_i\}$
  - $a_i \in T_i$ iff $c_i \notin T_i$
  - Let t be a_i or c_i, then for every other operation p_i: $p_i \underset{i}{<} t_i$
  - If $r_i[x]$ and $w_i[x] \in T_i$, then either $r_i[x] \underset{i}{<} w_i[x]$ or $w_i[x] \underset{i}{<} r_i[x]$

- Transactions are often represented as directed acyclic graphs (DAGs):



- $r_2[x] <_2 w_2[z], w_2[z] <_2 c_2, r_2[x] <_2 c_2, r_2[y] <_2 w_2[z], r_2[y] <_2 c_2$
- Transitive relationships are implicitly contained in the graph

Histories (Schedules)

- Multiple transactions can be executed concurrently
- This is described by a *history* (schedule)
- A history specifies how operations from different transactions are executed relative to each other
- Since different operations can be executed in parallel, a history is a partial order
-

Conflicting Operations

- Operations that conflict with each other must not be executed in parallel
- Two operations conflict with each other if both access the same data item and at least one of them is a write operation

	T_i	
T_j	$r_i[x]$	$w_i[x]$
$r_j[x]$		¬
$w_j[x]$	¬	¬

Definition of Histories

- Let $T = \{T_1, T_2, ..., T_n\}$ be a set of transactions
- A history H over T is a partial order with the ordering relation $\underset{H}{<}$, such that
  - $H = \bigcup_{i=1}^{n} T_i$
  - $\underset{H}{<} \supseteq \bigcup_{i=1}^{n} \underset{i}{<}$
  - For any two operations $p, q \in H$ that conflict with each other: either $p \underset{H}{<} q$ or $q \underset{H}{<} p$

Example of a History

$$
\begin{array}{llllll}
 & & r_2[x] \rightarrow & w_2[y] \rightarrow & w_2[z] \rightarrow & c_2 \\
 & & \uparrow & \uparrow & \uparrow & \\
H = & r_3[y] \rightarrow & w_3[x] \rightarrow & w_3[y] \rightarrow & w_3[z] \rightarrow & c_3 \\
 & & \uparrow & & & \\
 & r_1[x] \rightarrow & w_1[x] \rightarrow & c_1 & &
\end{array}
$$

(Conflict) Equivalence

- Two histories H and H' are *(conflict) equivalent* $(H \equiv H')$ if:
  - They contain the same sets of transactions (including all their operations)
  - They order the conflicting operations of the non-aborted transactions in the same way
- The idea is not to change the computed final result

Example

$$
\begin{aligned}
& r_1[x] \rightarrow w_1[y] \rightarrow r_2[z] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\
\equiv\ & r_1[x] \rightarrow r_2[z] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\
\equiv\ & r_2[z] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[y] \rightarrow c_2 \rightarrow c_1 \\
\not\equiv\ & r_2[z] \rightarrow r_1[x] \rightarrow w_2[y] \rightarrow w_1[y] \rightarrow c_2 \rightarrow c_1
\end{aligned}
$$

Serializability

- Since serial histories are safe, it is desirable to have histories with similar properties
- In particular, one wants a history that is equivalent to a serial history
- Such a history is called *serializable*

Serializability(2)

- Precise definition:
  - The *committed projection* $C(H)$ of a history H contains only the successfully committed transactions
  - A history H is serializable if $C(H)$ is equivalent to a serial history H_s

Serializability Theorem

- How can serializability be checked?
- A history is serializable if and only if its *serializability graph* $SG(H)$ is acyclic

Serializability Graph

- The serializability graph $SG(H)$ of a history $H = \{T_1, ..., T_n\}$ is a directed graph with the following properties:
  - The nodes are the successfully committed transactions of H
  - An edge from T_i to T_j is added if there are two conflicting operations p_i and q_j with $p_i \underset{H}{<} q_j$

Example

- History H

$$H = w_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow r_3[y] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_3[y] \rightarrow c_3$$

- $SG(H)$

$$SG(H) = \begin{array}{l} T_1 \nearrow T_2 \\ \qquad \searrow T_3 \end{array}$$

Example(2)

- H is serializable
- Possible orders

$$H_s^1 = T_1 \mid T_2 \mid T_3$$
$$H_s^2 = T_1 \mid T_3 \mid T_2$$
$$H \equiv H_s^1 \equiv H_s^2$$

Example(3)

$$
H = \begin{array}{ccccccc}
 & r_1[x] & \rightarrow & w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 \\
 & & & \uparrow & & \uparrow & & \\
 & & \swarrow & r_2[x] & \rightarrow & w_2[y] & \rightarrow & c_2 \\
 & & & \downarrow & & & & \\
 & r_3[x] & \rightarrow & w_3[x] & \rightarrow & c_3 & &
\end{array}
$$

$$
SG(H) = \begin{array}{ccc}
 & & T_3 \\
 & \nearrow & \\
T_2 & & \uparrow \\
 & \searrow & \\
 & & T_1
\end{array}
$$

Example(4)

- H is serializable
- Possible order

$$H_s^1 = T_2 \mid T_1 \mid T_3$$

$$H \equiv H_s^1$$

Example(5)

$$
\begin{array}{rccccc}
 & w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 \\
H= & \uparrow & & \downarrow & & \\
 & r_2[x] & \rightarrow & w_2[y] & \rightarrow & c_2 \\
SG(H)= & T_1 & \leftrightarrows & T_2 & &
\end{array}
$$

- H is not serializable

Further Properties

- Further properties are desirable for transactions:
  - Recoverability
  - Avoiding cascading aborts (ACA)
  - Strictness

Further Properties(2)

- First we have to define write/read dependencies (reads-from relationship)
- A transaction T_i reads (data item x) from transaction T_j if
  - $w_j[x] < r_i[x]$
  - $a_j \not< r_i[x]$
  - If there is a $w_k[x]$ with $w_j[x] < w_k[x] < r_i[x]$, then $a_k < r_i[x]$
- A transaction can also read from itself

Recoverability

- A history is *recoverable* if the following holds
  - Whenever a transaction T_i reads from another transaction T_j ($i \neq j$) and $c_i \in H$, then $c_j < c_i$
- The transactions must adhere to a certain commit order
- With non-recoverable histories, problems with the C and D of the ACID properties can arise

Recoverability(2)

$$H = w_1[x]r_2[x]w_2[y]c_2a_1$$

- H is not recoverable
- The consequences are:
  - If the result of T_2 is kept as is, we have inconsistent data (T_2 has read data from an aborted transaction)
  - If we roll back T_2, we undo the changes of a committed transaction

Cascading Aborts

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 (**abort**)				

Cascading Aborts(2)

- A history *avoids cascading aborts* if the following holds
  - Whenever a transaction T_i reads from another transaction T_j ($i \neq j$), then $c_j < r_i[x]$
- Reading is only allowed from already committed transactions

Strictness

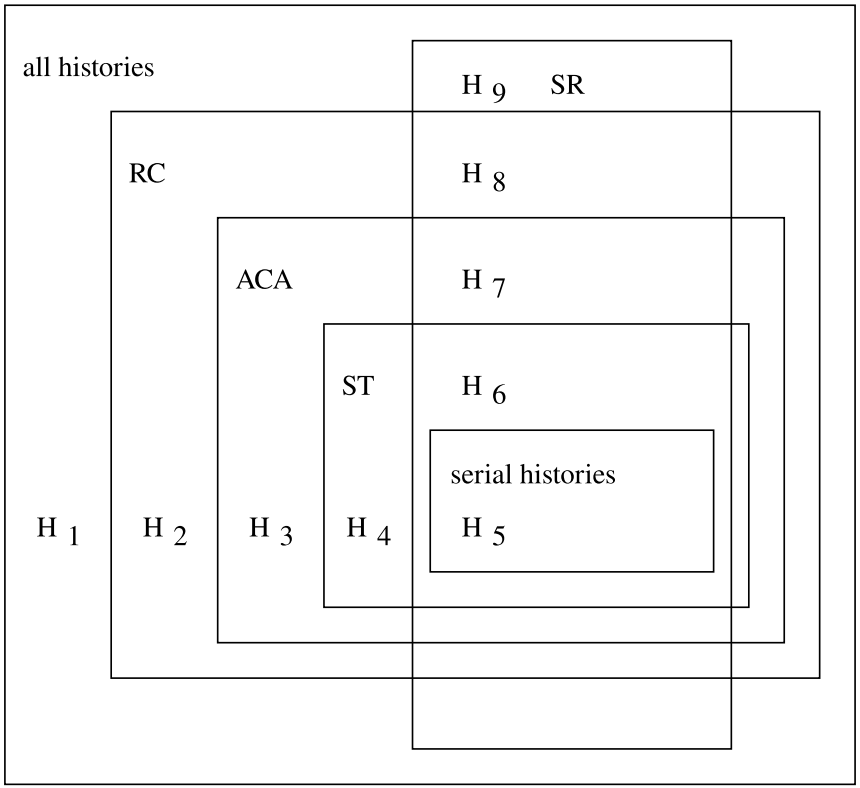
- A history is *strict* if the following holds
  - For two operations $w_j[x] < o_i[x]$ (with $o_i[x] = r_i[x]$ or $w_i[x]$) either $a_j < o_i[x]$ or $c_j < o_i[x]$
- Data items may only be read or overwritten if written by already completed (committed or aborted) transactions

- Physical logging may only be used for recovery with strict histories

$$
\begin{array}{ll}
x = 0 & \\
w_1[x, 1] & \text{before image of } T_1\text{: } 0 \\
x = 1 & \\
w_2[x, 2] & \text{before image of } T_2\text{: } 1 \\
x = 2 & \\
a_1 & \\
c_2 &
\end{array}
$$

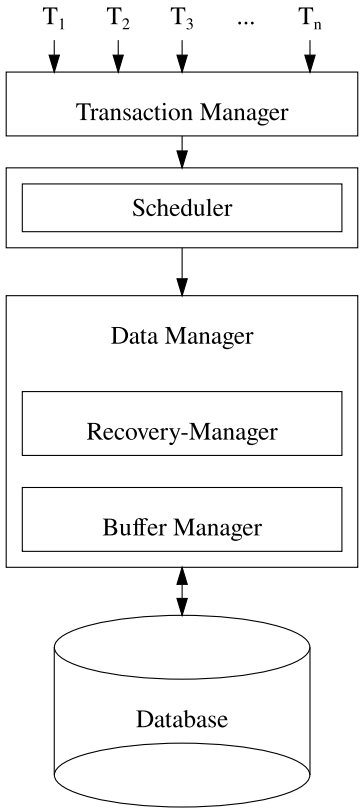
When T_1 aborts, x is incorrectly set to 0

Classification



SR: serializable, RC: recoverable, ACA: avoids cascading aborts, ST: strict

Database Scheduler



Database Scheduler(2)

- A *scheduler* is a program that orders the incoming operations and ensures a serializable and recoverable history
- Several options after receiving an operation:
  - Execute (immediately)
  - Reject
  - Delay

Database Scheduler(3)

- There are two broad strategies:
  - Pessimistic
  - Optimistic

Pessimistic Schedulers

- Scheduler delays received operations
- When several operations are available, the scheduler determines an order that is as clever as possible
- Most important representative: lock-based scheduler (widely used in practice)

Optimistic Schedulers

- Scheduler sends received operations to execution as quickly as possible
- May have to repair “damage” later
- Most important representative: timestamp-based scheduler

- Main idea is relatively simple:
  - Each data item has an associated lock
  - Before a transaction T_i may access it, it must request the lock
  - If another transaction T_j holds the lock, T_i does not get the lock and must wait until T_j has released the lock
  - Only one transaction can hold the lock and access the data item
- How can serializability be guaranteed?

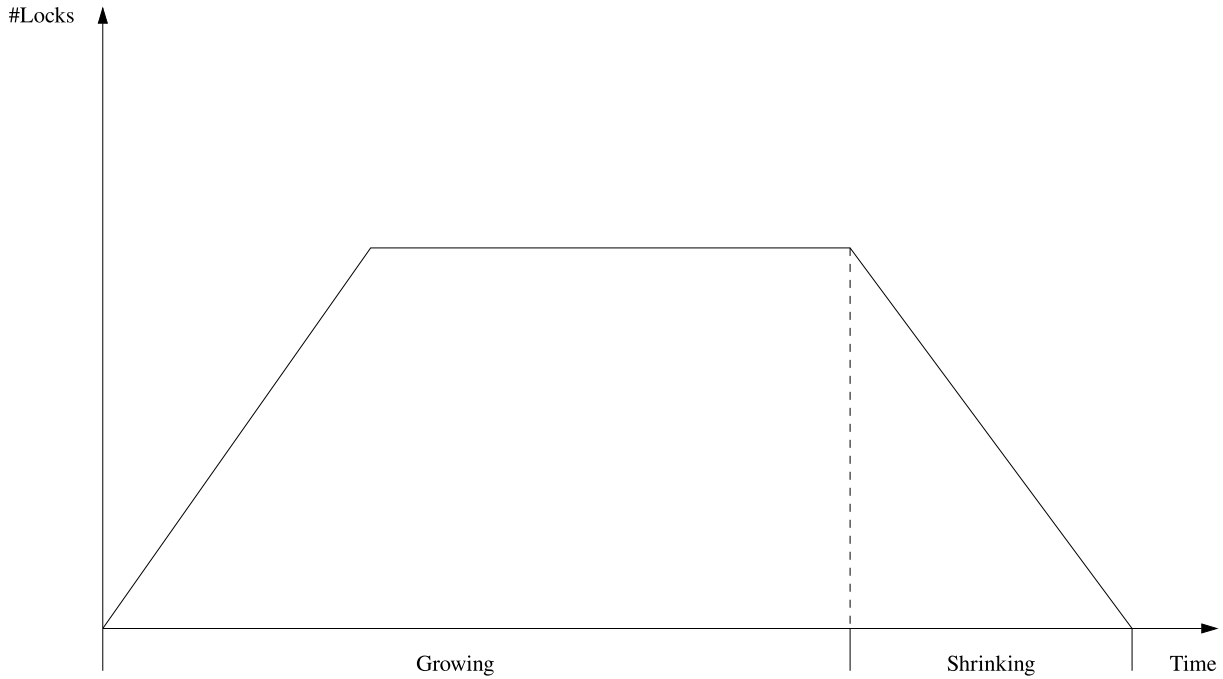
- Abbreviated as 2PL
- Two lock modes:
  - S (shared, read lock)
  - X (exclusive, write lock)
  - Compatibility matrix:

	held lock		
requested lock	none	S	X
S	√	√	–
X	√	–	–

Definition

- Every object that is to be used by a transaction must first be locked accordingly
- A transaction cannot request a lock it already holds again
- If a lock cannot be granted (according to the matrix), the transaction is put into a wait queue
- After the first release of a lock, a transaction may not request any further locks (there are two phases)
- At the end of the transaction, a transaction must release all locks

Two Phases



- Growing phase: locks are acquired, but none are released
- Shrinking phase: locks are released, but none are acquired

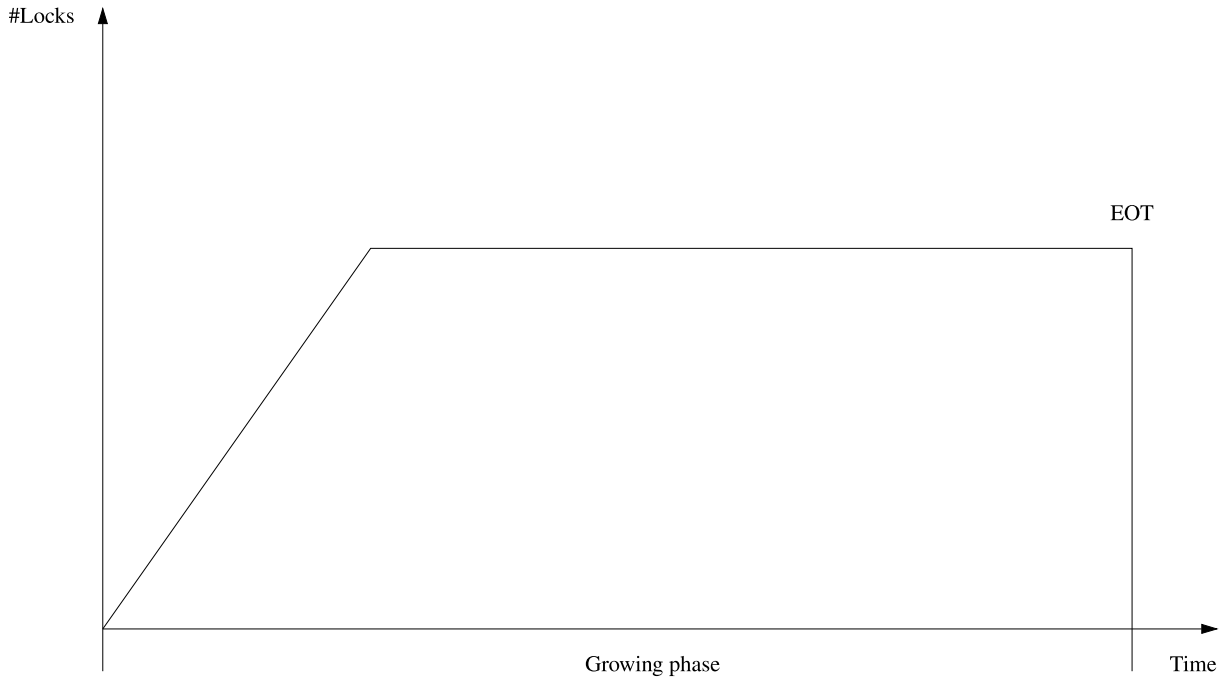
Interleaving under 2PL

Step	T_1	T_2	Remark
1.	**BOT**		
2.	**lockX**$[x]$		
3.	$r[x]$		
4.	$w[x]$		
5.		**BOT**	
6.		**lockS**$[x]$	T_2 must wait
7.	**lockX**$[y]$		
8.	$r[y]$		
9.	**unlockX**$[x]$		wake up T_2
10.		$r[x]$	
11.		**lockS**$[y]$	T_2 must wait
12.	$w[y]$		
13.	**unlockX**$[y]$		wake up T_2
14.		$r[y]$	
15.	**commit**		
16.		**unlockS**$[x]$	
17.		**unlockS**$[y]$	
18.		**commit**	

Strict 2PL

- 2PL does not rule out cascading aborts
- Extension to *strict* 2PL:
  - all locks are held until the end of the transaction
  - thus, cascading aborts are ruled out (the generated schedules are even strict)

Strict 2PL(2)



- Example of a deadlock:

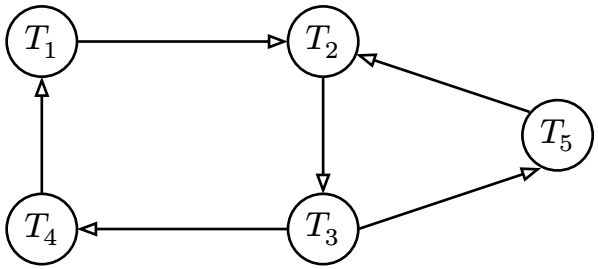
T_1	T_2
bot	
$\text{lockX}_1(a)$	
$w_1(a)$	
$\hookrightarrow$	bot
	$\text{lockS}_2(b)$
	$r_2(b)$
$\text{lockX}_1(b)$	$\hookleftarrow$
$\hookrightarrow$	$\text{lockS}_2(a)$

Detecting Deadlocks

- No transaction should wait “forever” for a lock
- One strategy for detecting deadlocks is time-out
  - Finding the right duration is problematic
- Precise method uses wait-for graphs
  - Nodes are transactions, edges are waits-for relationships
  - If the graph contains cycles, there is a deadlock

Wait-For Graph

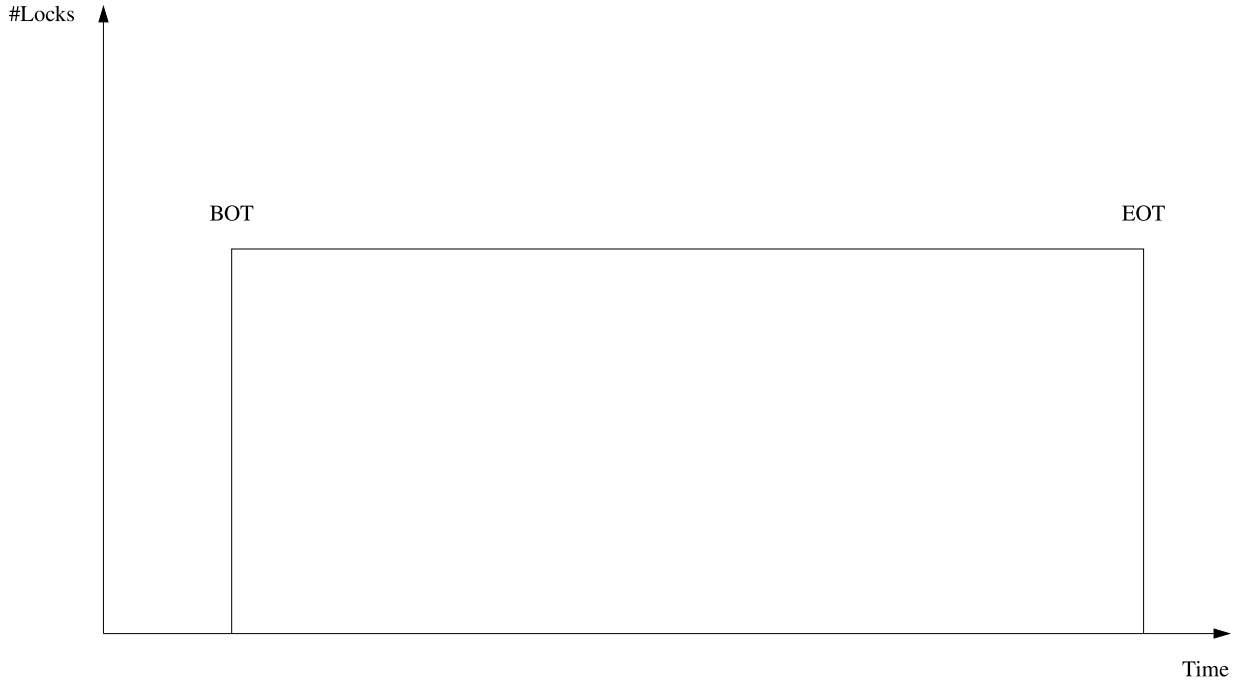
- Example



- Wait-for graph has cycles, i.e., there is a deadlock
- Here, the cycles can be resolved by rolling back T_2 or T_3

Deadlock Avoidance

- Deadlocks can be avoided by *preclaiming*
- Preclaiming means that all locks are requested at the beginning of a transaction
- Unrealistic in practice



Deadlock Avoidance(2)

- Time-out is often too cautious, sometimes transactions are aborted when there is only the suspicion of a deadlock
- Another method is to decide at the moment when T_i requests a lock that is held by T_j
  - Transactions are assigned priorities
  - If the priority of T_i is higher, T_i may wait
  - If the priority of T_i is lower, T_i is aborted

Deadlock Avoidance(3)

- Priorities prevent waiting from leading to a deadlock
- Priorities must be assigned carefully
  - If an aborted transaction T_i always gets low priorities when restarted, transactions with higher priorities can always "cut in line"
  - T_i never gets its turn, we have no deadlock, but a *livelock*

Deadlock Avoidance(4)

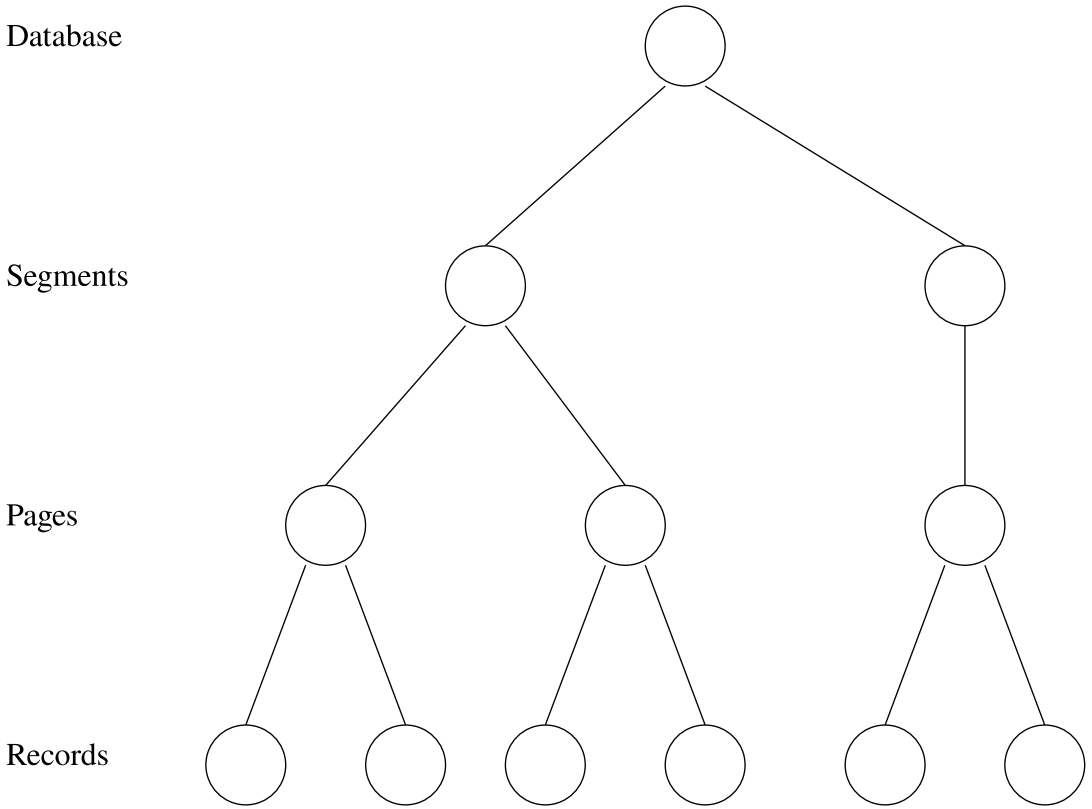
- Avoiding deadlocks and livelocks: using timestamps as priorities
- Timestamps are unique and increase monotonically over time
- A transaction is assigned a timestamp ts when it first starts, on restart it keeps its old timestamp
- The older the timestamp, the higher the priority
- Eventually, a repeatedly aborted transaction has the oldest timestamp, livelocks are prevented

Deadlock Avoidance(5)

- Assume T_j holds a lock, T_i requests it
- With timestamps, a scheduler can now follow two different strategies
  - Wait-Die:
    If $ts(T_i) < ts(T_j)$,
    then T_i waits, otherwise T_i aborts
  - Wound-Wait:
    If $ts(T_i) < ts(T_j)$,
    then T_j aborts, otherwise T_i waits

Phantom Problem

- With (strict) 2PL we have solved all problems mentioned at the beginning of the chapter, except for the phantom problem
- The phantom problem cannot be solved with locks on data items, since no locks can be requested on non-existent data items
- Solution of the problem by *hierarchical lock granularities* (multi-granularity locking: MGL)


Database
Segments
Pages
Records

Extended Lock Modes for MGL

- S (shared): for readers
- X (exclusive): for writers
- IS (intention share): for intended reading further down in the hierarchy
- IX (intention exclusive): for intended writing further down in the hierarchy

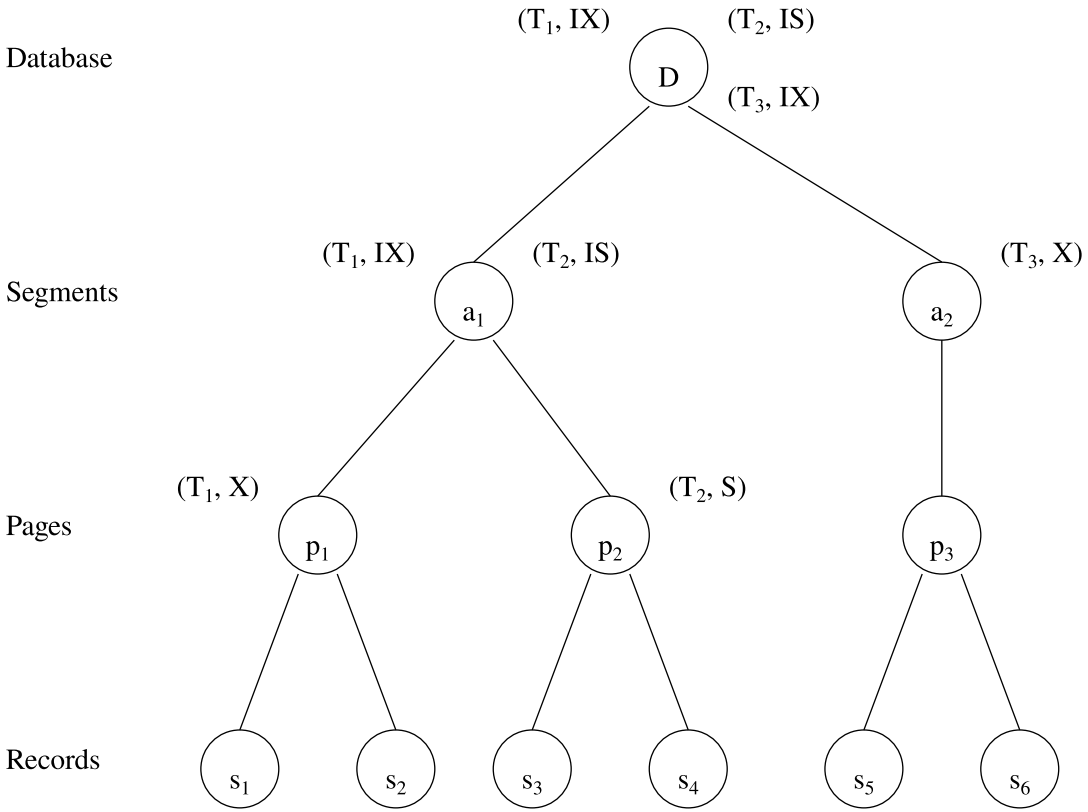
Compatibility Matrix

	held lock				
requested	none	*S*	*X*	*IS*	IX
S	√	√	–	√	–
X	√	–	–	–	–
IS	√	√	–	√	√
IX	√	–	–	√	√

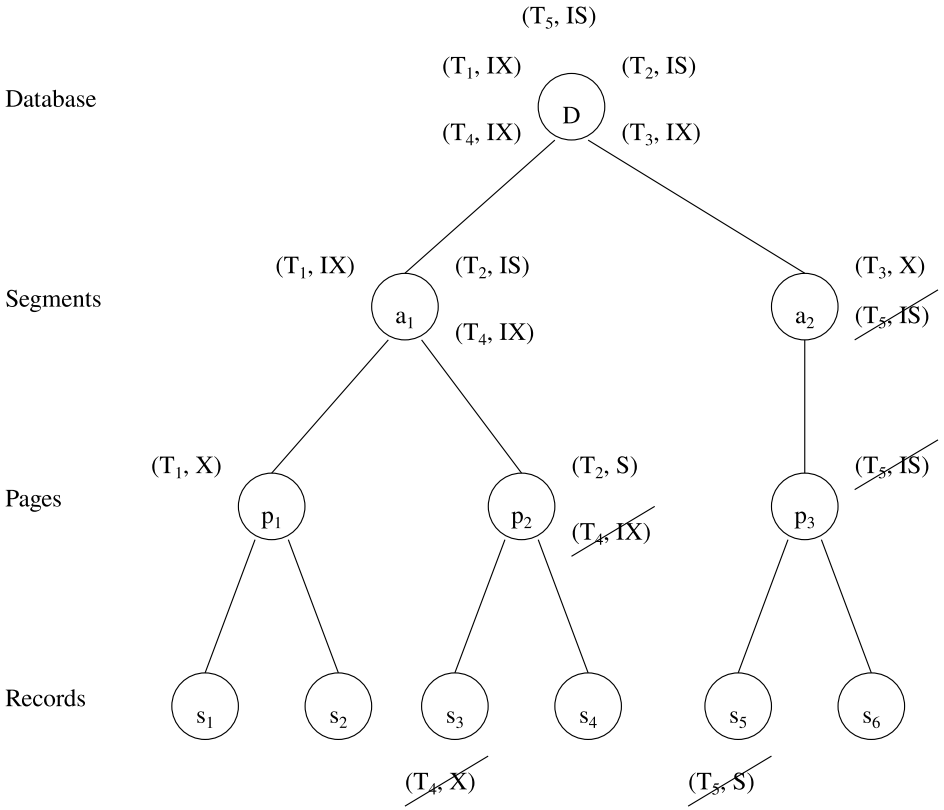
Locking Protocol

- Locks are requested top-down in the hierarchy
  - For an S or IS lock, all ancestors in the hierarchy must be locked in IS or IX mode
  - For an X or IX lock, all ancestors in the hierarchy must be held in IX mode
- Locks are released bottom-up (a lock is only released if no lock is held anymore on any descendant of the node)

Example


Database
Segments
Pages
Records
(T1, IX)
(T2, IS)
(T3, IX)
D
(T1, IX)
(T2, IS)
(T3, X)
a1
a2
(T1, X)
(T2, S)
p1
p2
p3
s1
s2
s3
s4
s5
s6

Example(2)


Database
Segments
Pages
Records
(T5, IS)
(T1, IX)
(T2, IS)
D
(T4, IX)
(T3, IX)
(T1, IX)
(T2, IS)
(T3, X)
a1
a2
(T4, IX)
(T5, IS)
(T1, X)
(T2, S)
(T5, IS)
p1
p2
p3
(T4, IX)
s1
s2
s3
s4
s5
s6
(T4, X)
(T5, S)

Example(3)

- Transactions T_4 and T_5 are blocked
- No deadlock here yet, but deadlocks are also possible in the simple MGL protocol

- Besides the lock-based protocols there is another large class of protocols, timestamp-based synchronization
- The transaction manager assigns a unique timestamp to each transaction
- Each operation of the transaction gets this timestamp

- A scheduler uses the timestamps to order conflicting operations:
  - Assume $p_i[x]$ and $q_j[x]$ conflict with each other
  - $p_i[x]$ is executed before $q_j[x]$ iff the timestamp of T_i is older than the timestamp of T_j

Timestamps(2)

- For each data item x, the scheduler stores the timestamp of the last operation executed on x
- This is done for each operation type q: max-q-scheduled(x)
- When the scheduler receives an operation p, its timestamp is compared with all max-q-scheduled(x) with which p conflicts
- If the timestamp of p is older than any max-q-scheduled(x), p is rejected (and the transaction aborted)
- Otherwise, p is executed and max-p-scheduled(x) is updated

Further Properties

- The simple timestamp protocol may generate non-recoverable schedules
- Recoverability can be guaranteed by having transactions commit in timestamp order
- As long as transactions are still running from which a transaction T_i has read, a commit of T_i is delayed

Problems of Timestamps

- Timestamp-based synchronization is rarely used in practice
- Phantom problem is not solved
- Every operation effectively becomes a write operation, since the max-q-scheduled(x) fields always have to be updated

Summary

- Concurrency control is one of the most important functions of a DBMS
- Usually, it is hidden from the users, but by setting the isolation level one can influence the quality of this synchronization
- The two best-known approaches:
  - Lock-based synchronization
  - Timestamp-based synchronization