IN3020/4020 – Database Systems
Spring 2020, Week 10.1

Serialization and Concurrency Control
Part 1

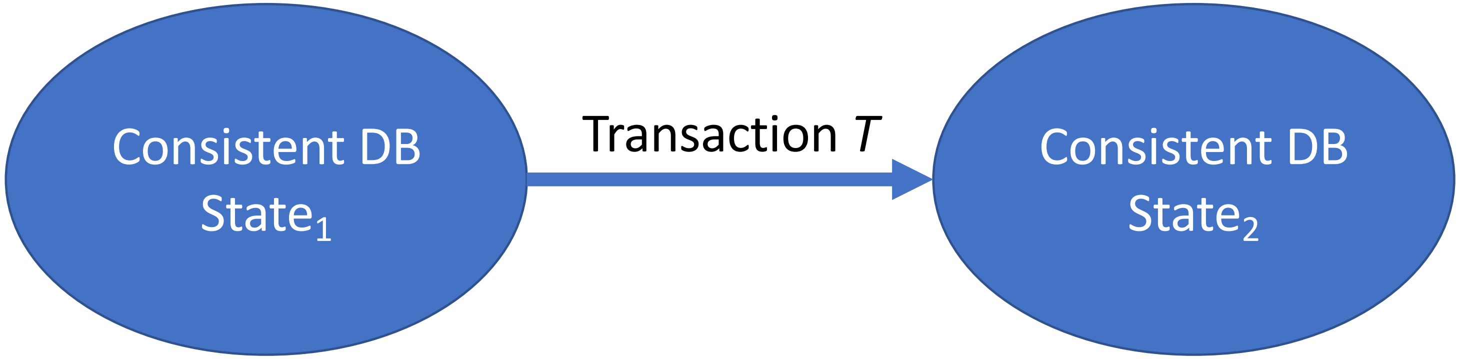
Dr. M. Naci Akkøk, Chief Architect, Oracle Nordics

Based upon slides by E. Thorstensen from Spring 2019

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

What is a transaction?

- A ***transaction*** is a sequence of operations that maintains ***consistency*** in the database



ACID characteristics (repetition)

- **A – Atomicity**
  Either the entire transaction is executed or none of it is executed.
- **C - Consistency preservation**
  Transactions shall maintain (data) consistency, as in the definition of transaction
- **I - Isolation**
  Transactions shall occur in isolation. No transaction will be affected by other transacstions
- **D - Durability** (permanency)
  Once transactions are completed, their effect should be lasting and not be affected by system failures

Ensuring/implementing ACID

- **Atomicity:** A transaction is an atomic processing unit (sinmgle unit of logic, single unit of work)
- The database system's recovery method (recovery manager) ensures **A** by reversing any changes a failed transaction *T* has made to the database
- This is done by reading the log and writing back the old values of data that *T* has changed
- (Database system´s concurrency control - transaction manager - cooperates with log manager and buffer manager for logging of transactions)

Ensuring/implementing ACID

- Consistency preservation: Transactions bring database from one consistent state to another (next) consistent state
- C is partially guaranteed by the database management system by ensuring that certain types of integrity rules are not violated (e.g., primary and foreign keys)
- If the DBMS cannot handle a rule, the database programmer must take responsibility for maintaining consistency

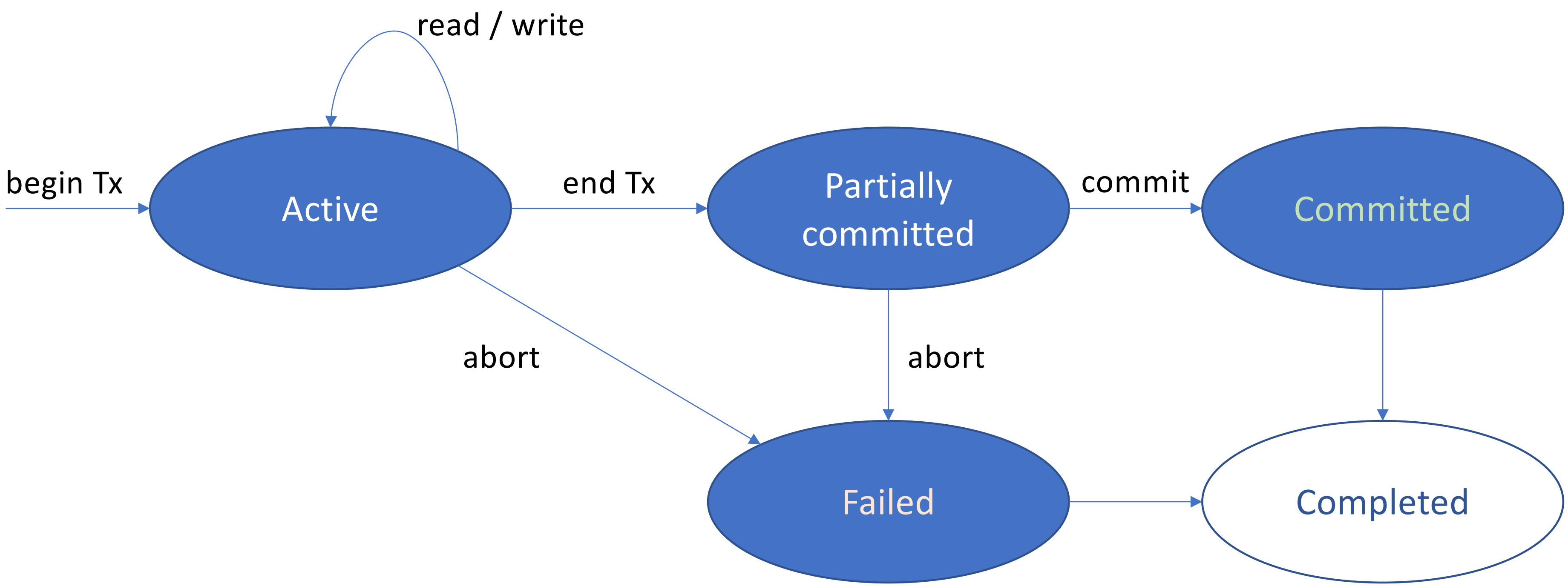
Ensuring/implementing ACID

- Isolation: Updates should not be visible to other transactions before the transaction is committed
- The concurrency control component (the scheduler - a subcomponent of transaction manager) is responsible for ensuring Isolation
- There are many ways to do this, and Isolation is one of the most difficult to enforce ACID features

Ensuring/implementing ACID

- Durability / permanency: Updates that are committed are permanent!
- The database system recovery method (the recovery manager) is responsible for ensuring Durability
- After a system crash, the log is read and data written by committed transactions are restored in the database, while transactions that have not committed are «reversed»

State transition diagram for executing transactions



We will look at challenges with f. ex. isolation

For that, we need several mechanisms, like **Serialization**

Next time…

Serializability

- The execution of a number of transactions is **serial** if the execution is completed completely for one transaction before the next transaction is executed.
- Execution serializable if the transaction executions are such that there exists a serial execution that gives the same total result
- Atomic execution of each transaction and serializable execution of a collection of transactions help ensure that the database remains consistent and that the application that initiated the transaction experiences the result as predictable (remember isolation).

Execution plans

- An **execution plan** (schedule) S for a set of transactions $\{T_1, ..., T_n\}$ is a merging of the operations in $T_1, ..., T_n$.

- Features/characteristics of S:
  - Each element in S is an operation in exactly one of the transactions
  - Each operation in a transaction is the element of S exactly once
  - S maintains the order of operations from each transaction

Example transactions

Integrity Rule: $A = B$

T1:	Read(A);	T2:	Read(A);
	$A \leftarrow A+100$;		$A \leftarrow A\times 2$;
	Write(A);		Write(A);
	Read(B);		Read(B);
	$B \leftarrow B+100$;		$B \leftarrow B\times 2$;
	Write(B);		Write(B);

Example execution plan S_A (serial!)

T1	T2	A	B
		25	25
Read(A); A $\leftarrow$ A+100;			
Write(A);		125	
Read(B); B $\leftarrow$ B+100;			
Write(B);			125
	Read(A); A $\leftarrow$ A$\times$2;		
	Write(A);	250	
	Read(B); B $\leftarrow$ B$\times$2;		
	Write(B);		250
		250	250

Example execution plan S_B (also **serial!**)

T1	T2	A	B
		25	25
	Read(A); A ← A×2;		
	Write(A);	50	
	Read(B); B ← B×2;		
	Write(B);		50
Read(A); A ← A+100;			
Write(A);		150	
Read(B); B ← B+100;			
Write(B);			150
		150	150

Example execution plan S_C

T1	T2	A	B
		25	25
Read(A); A ← A+100;			
Write(A);		125	
	Read(A); A ← A×2;		
	Write(A);	250	
Read(B); B ← B+100;			
Write(B);			125
	Read(B); B ← B×2;		
	Write(B);		250
		250	250

Example execution plan S_D

T1	T2	A	B
		25	25
Read(A); $A \leftarrow A+100$;			
Write(A);		125	
	Read(A); $A \leftarrow A\times2$;		
	Write(A);	250	
	Read(B); $B \leftarrow B\times2$;		
	Write(B);		50
Read(B); $B \leftarrow B+100$;			
Write(B);			150
		250	150

Example execution plan S_E

T1	T2'	A	B
		25	25
Read(A); A ← A+100;			
Write(A);		125	
	Read(A); A ← A×1;		
	Write(A);	125	
	Read(B); B ← B×1;		
	Write(B);		25
Read(B); B ← B+100;			
Write(B);			125
		125	125

NOTE: “Same” plan as S_D, but with new T2´

“Good” execution plans

- We want execution plans that are “good”
- The term “good” should be independent of
  - The initial state
  - Transaction semantics
- The term should only depend on the reading and writing operations and their order
- There are several possible definitions of «good».
  The purpose is to guarantee serializable executions!
- We will first look at "conflict serialization"

Some necessary concepts first

- **Transaction:** A sequence of read operations $r_i(A)$ and write operations $w_i(B)$
- **Transaction execution plan $\{T_1, ..., T_n\}$:** A merge of $T_1, ..., T_n$
- **Serial execution plan:** Plan where all operations in one transaction are completed before the next transaction is started
- **Conflict in an execution plan:**
  1. Read-write conflict: A couple of operations of the form $...r_i(A) ...w_k(A) ...$ or $...w_i(A) ...r_k(A) ...$ (where $i \neq k$)
  2. Write-write conflict: A couple of operations of the form $...w_i(A) ...w_k(A) ...$ (where $i \neq k$)
  3. Intra-transaction conflict: A pair of form operations $...o_i(A) ...o_i(B) ...$ where o_i is w_i or r_i

Conflict serializability

- Two execution plans S_1 and S_2 are called conflict equivalents if S_1 can be transformed into S_2 by a series of exchanges of neighboring operations that do not conflict with each other
- An execution plan is conflict serializable if it is conflict equivalent to a serial execution plan

Example

$S_C = r_1(A);w_1(A);r_2(A);w_2(A);r_1(B);w_1(B);r_2(B);w_2(B)$ 1

$r_1(A);w_1(A);r_2(A);r_1(B);w_2(A);w_1(B);r_2(B);w_2(B)$ 2

$r_1(A);w_1(A);r_1(B);r_2(A);w_2(A);w_1(B);r_2(B);w_2(B)$ 3

$r_1(A);w_1(A);r_1(B);r_2(A);w_1(B);w_2(A);r_2(B);w_2(B)$ 4

$\underbrace{r_1(A);w_1(A);r_1(B);w_1(B)}_{T1};\underbrace{r_2(A);w_2(A);r_2(B);w_2(B)}_{T2}$ 5

1. Read-write conflict: A couple of operations of the form $...r_i(A)\ ...w_k(A)\ ...$ or $...w_i(A)\ ...r_k(A)\ ...$ (where $i \neq k$)
2. Write-write conflict: A couple of operations of the form $...w_i(A)\ ...w_k(A)\ ...$ (where $i \neq k$)
3. Intra-transaction conflict: A pair of form operations $...o_i(A)\ ...o_i(B)\ ...$ where o_i is w_i or r_i

S_C can be transformed into a serial execution plan:
Thus, S_C is **conflict serializable**.

A “bad” execution plan

1. Read-write conflict: A couple of operations of the form $\ldots r_i(A) \ldots w_k(A) \ldots$ or $\ldots w_i(A) \ldots r_k(A) \ldots$ (where $i \neq k$)
2. Write-write conflict: A couple of operations of the form $\ldots w_i(A) \ldots w_k(A) \ldots$ (where $i \neq k$)
3. Intra-transaction conflict: A pair of form operations $\ldots o_i(A) \ldots o_i(B) \ldots$ where o_i is w_i or r_i

Let us look at the following execution plan (plan S_D):

$$S_D = r_1(A); w_1(A); r_2(A); w_2(A); r_2(B); w_2(B); r_1(B); w_1(B)$$

- We have a conflict between $w_1(A)$ and $r_2(A)$. These cannot switch place, so S_D cannot be conflict equivalent to the serial plan $T_2; T_1$
- We also have a conflict between $w_2(B)$ and $r_1(B)$, which also cannot change place. Thus, S_D cannot be conflict equivalent to the serial plan $T_1; T_2$ either
- Thus, S_D is <u>not</u> conflict serializable!

A “bad” execution plan (continued)

1. Read-write conflict: A couple of operations of the form ...$r_i(A)$ ...$w_k(A)$ ... or ...$w_i(A)$ ...$r_k(A)$ ... (where $i \neq k$)
2. Write-write conflict: A couple of operations of the form ...$w_i(A)$ ...$w_k(A)$ ... (where $i \neq k$)
3. Intra-transaction conflict: A pair of form operations ...$o_i(A)$ ...$o_i(B)$ ... where o_i is w_i or r_i

$$S_D = r_1(A);w_1(A);r_2(A);w_2(A);r_2(B);w_2(B);r_1(B);w_1(B)$$

All conflicts between operations in T_1 and T_2 are plotted

- The fact that T_1 must deal with A before T_2 does is called “T_1 has precedence over T_2 (on A)”, and we write it like this:
  $T_1 \rightarrow T_2$ in S_D
- But we also have that T_2 has precedence over T_1 (on B), so we have both $T_2 \rightarrow T_1$ and $T_1 \rightarrow T_2$ in S_D
- It is this kind of cyclic dependency that prevents S_D from being rearranged into a serial execution plan

A “bad” execution plan? Look at S_E

1. Read-write conflict: A couple of operations of the form $\ldots r_i(A) \ldots w_k(A) \ldots$ or $\ldots w_i(A) \ldots r_k(A) \ldots$ (where $i \neq k$)
2. Write-write conflict: A couple of operations of the form $\ldots w_i(A) \ldots w_k(A) \ldots$ (where $i \neq k$)
3. Intra-transaction conflict: A pair of form operations $\ldots o_i(A) \ldots o_i(B) \ldots$ where o_i is w_i or r_i

$$S_E = r_1(A);w_1(A);r_2(A);w_2(A);r_2(B);w_2(B);r_1(B);w_1(B)$$

Look at the plan – also slides 11 through 15:

- When the transaction semantics are abstracted away, we do not see the difference between S_D and S_E.
- We know that, unlike S_D, S_E is serializable (see slide 15 where the semantics of S_E transactions are described), but it still is <u>not</u> conflict serializable! Why was that? (See previous slide)

Conflict serializable ⇒ Serializable

- Any conflict serializable execution plan is serializable
  - Changing non-conflicting operations will not change the outcome of the execution
  - Thus, it is sufficient to allow only execution plans that are conflict serializable
- Observe and be aware: There are execution plans that are not conflict serializable but still serializable
- If we reject plans that are not conflict serializable, we may therefore be rejecting some plans that would have gone well, but <u>it is too expensive or impossible to check serializability in general</u>
- How can we check conflict serializability in practice?

Precedence graphs

- Let S be an execution plan, and let $p_i(A)$ and $q_k(B)$ be two (arbitrary) operations in S. The notation $p_i(A) <_S q_k(B)$ means that $p_i(A)$ is to be executed before $q_k(B)$ in S. Then the precedent graph of S, defined as P(S), is as follows:
  - Nodes: The transactions in S
  - Edges: The precedents in S
    - $T_i \rightarrow T_k$ (where $i \neq k$) if
      1. $p_i(A) <_S q_k(A)$ and
      2. at least one of p_i or q_k is a write operation

Exercise (group): Draw P(S) for S = $w_3(A); w_2(C); r_1(A); w_1(B); r_1(C); w_2(A); r_4(A); w_4(D)$

Note: There are 4 transactions (T_1, T_2, T_3, T_4), and not all data elements A, B, C and D are in every transaction. Is S serializable?

Reminder
S is an **execution plan** and P(S) is a **precedence graph** for the execution plan S

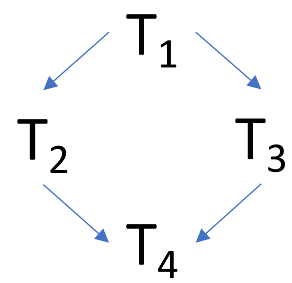
Precedence graphs - Lemma

- Lemma: S_1 and S_2 are conflict equivalent plans $\Longrightarrow$ $P(S_1) = P(S_2)$
- Proof: We show that $P(S_1) \neq P(S_2)$ $\Longrightarrow$ S_1 and S_2 are not conflict equivalent.
  - Suppose that S_1 and S_2 are both merging/interweaving of transactions $\{T_1, ..., T_n\}$, but that $P(S_1) \neq P(S_2)$.
  - Then i and k ($i \neq k$) exist such that $T_i \rightarrow T_k$ is an edge in $P(S_1)$, but not in $P(S_2)$.
  - This means that there are operations p_i and q_k that conflict with a data element A such that
    - $S_1 = ... p_i(A) ... q_k(A) ...$ (hence the edge $T_i \rightarrow T_k$ in $P(S_1)$)
    - $S_2 = ... q_k(A) ... p_i(A) ...$ (so there is also an edge $T_k \rightarrow$ Ti in $P(S_2)$)
  - This shows that S_1 and S_2 are not conflict equivalent.

Precedence graphs - continued

- Note: We cannot conclude the opposite, i.e., from $P(S_1) = P(S_2)$ that S_1 and S_2 are conflict equivalents.
- Proof (case example):
  - $S_1 = w_1(A); r_2(A); w_2(B); r_1(B)$
  - $S_2 = r_2(A); w_1(A); r_1(B); w_2(B)$
- S_1 and S_2 are obviously not conflict equivalent.
- But $P(S_1)$ and $P(S_2)$ both have the two nodes T_1 and T_2 and the two edges $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_1$, so $P(S_1) = P(S_2)$.

Precedence graphs - Theorem



- Theorem:
  $P(S)$ is acyclic $\Leftrightarrow$ S is conflict serializable
- Proof ($\Longrightarrow$)
  Suppose that $P(S)$ is acyclic. Restructure S as follows:
  1. Choose a transaction T_1 that has no incoming edges in $P(S)$
  2. Move all operations in T_1 to the start of S
     (in the order they occur in T_1), i.e., $S = \ldots.\ q_k(B)\ \ldots.\ p_1(A)\ \ldots.$
  3. Now we have S_1 = [the operations in T_1] [the rest of S>]
  4. Repeat 1-3 to serialize the rest of S.

Enforcement of serializability and serializability protocols

- Method 1:
  Run the system and register P(S)
  "At the end of the day" we check if P(S) is acyclic, i.e., if everything went well
- Method 2:
  Check in advance that the execution plan can never cause cycles in P(S)
- A framework that supports method 2 is called a serialization protocol

Locking protocols

- We introduce two new types of operation:
  - **Lock:** l_i(A) – T_i puts (an exclusive) lock on A
  - **Unlock:** u_i(A) – T_i releases the lock on A
- In addition, we require that DBMS must maintain a lock table that shows which data elements are locked by which transactions
- Most DBMS´ have their own lock manager modules that keep track of the lock table

Execution plan S_D with locks

T1	T2	A	B
		25	25
$l_1(A)$; Read(A)			
$A \leftarrow A+100$; Write(A); $u_1(A)$		125	
	$l_2(A)$; Read(A)		
	$A \leftarrow Ax2$; Write(A); $u_2(A)$	250	
	$l_2(B)$; Read(B)		
	$B \leftarrow Bx2$; Write(B); $u_2(B)$		50
$l_1(B)$; Read(B)			
$B \leftarrow B+100$; Write(B); $u_1(B)$			150
		250	150

Note that Locks alone do NOT guarantee serializability!

Locking rules – 2 Phase Locking (2PL)

- **Rule 1** - Well-formed transactions:
  Before T_i performs operation $p_i(A)$, T_i must have performed $l_i(A)$, and it should perform $u_i(A)$ after $p_i(A)$
  Example: T_i: ... $l_i(A)$ ... $r_i(A)$ ... $w_i(A)$ ... $u_i(A)$ ...
- **Rule 2** - Allowed ("Legal") Execution Plans: Execution plans cannot allow two transactions to lock on the same data element at the same time

  i.e.,: S: ... $l_i(A) \underbrace{\ldots\ldots\ldots\ldots}_{\text{No } l_k(A) \text{ (for } k \neq i)} u_i(A)$ ...

Locking rules – 2 Phase Locking (2PL)

- **Rule 3 –** 2 phase locking
- A transaction that has performed an unlock operation is not allowed to perform other lock operations
  $T_i = \underbrace{\ldots\ldots\ldots\ldots}_{\text{No } u_i(B)} l_i(A) \underbrace{\ldots\ldots}_{\text{No } l_i(B)} u_i(A) \ldots\ldots$
- The time leading up to the transaction's first unlock operation is called the transaction's <u>growing phase</u>
- The time from the transaction's first unlock operation is called the transaction <u>shrinking phase</u>

Conflict rules for lock/unlock

- $l_i(A)$, $l_k(A)$ leads to conflict
- $l_i(A)$, $u_k(A)$ leads to conflict

- Note that the following two situations do not lead to conflict:
  - $u_i(A)$, $u_k(A)$
  - $l_i(A)$, $r_k(A)$

Start of the shrinking phase

Reminder:
- The time leading up to the transaction's first unlock operation is called the transaction's <u>growing phase</u>
- The time from the transaction's first unlock operation is called the transaction <u>shrinking phase</u>

- **Rule 1:** Well formed transactions
- **Rule 2:** Allowed or "legal" execution plans
- **Rule 3:** 2 phase locking (2PL)

- A helping definition:
  $Sh(T_i)$ = first unlock operation that T_i performs
- **Lemma:** If $T_i \rightarrow T_k$ in P(S), then $Sh(T_i) <_S Sh(T_k)$
- **Proof:** Ti → Tk means that
  S = … $p_i(A)$ … $q_k(A)$ …; where p_i and q_k are in conflict
  - **Rule 1** states that $u_i(A)$ must come after $p_i(A)$ and $l_k(A)$ before $q_k(A)$
  - **Rule 2** states that $l_k(A)$ must come after $u_i(A)$.
    Thus we have S = … $p_i(A)$ … $u_i(A)$ … $l_k(A)$ … $q_k(A)$ …;
  - **Rule 3** states that $Sh(T_i)$ cannot come after $u_i(A)$ and that $Sh(T_k)$ must come after $l_k(A)$
- Q.E.D. We have proved that $Sh(T_i)$ must come before $Sh(T_k)$ in S

2PL ensures conflict seriazability

- **THEOREM:** If a plan S complies with rules 1, 2 and 3,
  then S is conflict serializable

- **Proof:** According to the earlier theorem (see slides 25 & 28, as well as 32 through 35), it is sufficient to show that if a plan S complies with rules 1, 2 and 3, then the precedence graph P(S) is acyclic
- Thus, assume (ad absurdum) that P(S) has a cycle T1 → T2 → ... → Tn → T1
- According to the lemma, then

  $$Sh(T_1) <_S Sh(T_2) <_S \dots <_S Sh(T_n) <_S Sh(T_1)$$

- But this is impossible, so P(S) is acyclic!

Reminder:
$Sh(T_i)$ = first unlock operation that T_i performs

Deadlock

T1	T2
$l_1(A)$; Read(A);	
$A \leftarrow A+100$; Write(A);	
	$l_2(B)$; Read(B);
	$B \leftarrow B\times 2$; Write(B);
$l_1(B)$; Must wait for T2	
	$l_2(A)$; Must wait for T1

This demonstrates that 2PL is NOT a guarantee against deadlock!

Read and write locks

- For improved concurrency, we can use two different types of locks:
  - **Shared lock** (**sl**) that allows other transactions to read the data element but not write it
  - **Write lock** (eXclusive lock, **xl**) that does not allow other transactions to read or write the data element
- Notation:
  - $sl_k(A)$: T_k puts a read lock (shared lock) on A
  - $xl_k(A)$: T_k puts a write lock (eXclusive lock) on A
  - $u_k(A)$: T_k deletes its lock(s) on A (both read and write)
- $sl_k(A)$ is not executed if any transaction other than T_k has write lock on A
- $xl_k(A)$ is not executed if a transaction other than T_k has locked A (it does not matter whether it is a read or write lock)

Rules for read and write locks

- For **Well-formed transactions** (rule 1)
  Any transaction T_k must comply with the following three rules:
  - An $r_k(A)$ must come after an $sl_k(A)$ or $xl_k(A)$ without any $u_k(A)$ in between
  - A $w_k(A)$ must come after an $xl_k(A)$ without any $u_k(A)$ in between
  - There must be a $u_k(A)$ after an $sl_k(A)$ or $xl_k(A)$
- For **2-phase lock** (rule 3)
  In addition, any 2PL transaction T_k must comply with the following:
  - No $sl_k(A)$ or $xl_k(A)$ can come after an $u_k(B)$ regardless of what A and B are

Rules for read and write locks (continued)

- **Allowed execution plans** (rule 2)
  Each data element is either unlocked, or has one write lock, or has one or more read locks.
  This is ensured by all plans S following these rules:
  - If $xl_i(A)$ occurs in S, it must be followed by a $u_i(A)$ before an $xl_k(A)$ or $sl_k(A)$ with $k \neq i$
  - If $sl_i(A)$ occurs in S, it must be followed by a $u_i(A)$ before there can be an $xl_k(A)$, where $k \neq i$

Conflict serializability of SL / XL plans

THEOREM: If a plan S complies with the rules for read and write locks on the two previous slides, then S is conflict serializable.

PROOF: Almost identical to the **proof that plans using only exclusive locks ensure conflict serializability**, but with the only difference being that we need that neither

$sl_i(A)$ followed by $sl_k(A)$

nor

$sl_i(A)$ followed by $u_k(A)$

is a conflict.

Compatibility matrices

- Compatibility matrices are used to store the lock allocation rules when using multiple lock types
- The matrices have one row and one column for each lock type
- Compatibility matrices are interpreted as follows:
  - If T_i asks to put a type K lock on data element A, it only gets it if there is a 'Yes' in column K in all rows R of the matrix where some other T_k has a type R lock on A
- Example: Compatibility matrix for S / X locks

Lock that T is asking to get on A ↓

Lock that A has ↓	S	X
S	Yes	No
X	No	No

Upgrading the locks

- For better (more efficient) concurrency, we can allow T to first set read lock and then upgrade it to write lock if needed
- Example:

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$sl_1(B)$; $r_1(B)$;	
$xl_1(B)$; **Rejected!**	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_1(B)$;	

Upgrading the locks (continued)

- One disadvantage is that upgrading locks increases the risk of deadlock!
- Example:

T_1	T_2
$sl_1(A); r_1(A);$	
	$sl_2(A); r_2(A);$
$xl_1(A);$ **Rejected!**	
	$xl_2(A);$ **Rejected!**

- The example illustrates that protocols that use lock upgrades are only suitable if there are many more read than write transactions

Update locks

- An update lock is a read lock that will later be upgraded to a write lock
- Update locks are denoted by U (Update lock)
- The compatibility matrix for S / X / U locks comes in two variants (where the asymmetric is most common):
  - an asymmetric ('N') that prioritizes writing transactions
  - a symmetric ('J') that prioritizes reading transactions

Requests lock ↓

Has lock ↓	S	X	U
S	Yes	No	Yes
X	No	No	No
U	Yes/No	No	No

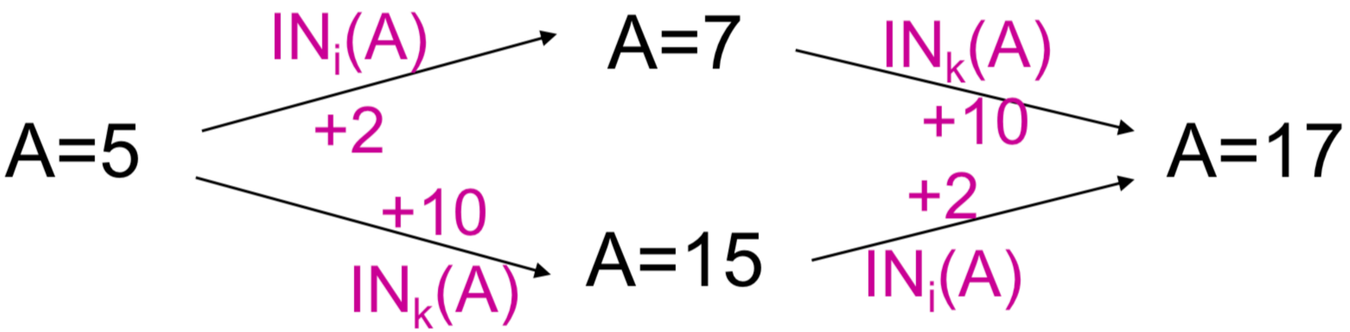
Update locks (continued)

- Plans that earlier resulted in deadlock due to read-to-write lock upgrades do not do so with the use of update locks.
(BUT **NOTE!** There may be other causes of deadlock!)
- Example (which was a deadlock earlier):

T_1	T_2
$ul_1(A)$; $r_1(A)$;	
	$ul_2(A)$; **Rejected!**
$xl_1(A)$; $w_1(A)$; $u_1(A)$;	
	$ul_2(A)$; $r_2(A)$;
	$xl_2(A)$; $w_2(A)$; $u_2(A)$;

Incremental locks

- Atomic increment operation: $IN_i(A)$
  {Read (A); A ← A + v; Write (A)}
- $IN_i(A)$ and $IN_k(A)$ are not in conflict!



Incremental locks (continued)

- The purpose is to streamline bookkeeping transactions
- Increment locks are denoted by I (looks sure like l, so be careful!)
- Increment locks conflict with both read and write locks, but not with other increment locks
- Here is the compatibility matrix for S / X / I locks:

	Requests lock ↓		
Has lock ↓	**S**	**X**	**I**
S	Yes	No	No
X	No	No	No
I	No	No	Yes

We continue with the challenges of concurrency…

For that, we will be looking at

new types of locks (alerts), lock management

and then **isolation**

Next time…

Lock scheduling

- In practice, no DBMS will allow the transactions to set or release any locks themselves
- Transactions perform only the read, write, commit and abort operations, and <u>optionally</u> update and increment
- The locks are entered into the transactions and are set and released by a separate module in DBMS called the lock manager (Lock Scheduler)
- Lock manager uses its own internal data structure, the lock table, to manage the locks
- The lock table is not part of the buffer area; (depending upon the DBMS) it is (usually) unavailable for the transactions

Lock scheduling, lock management

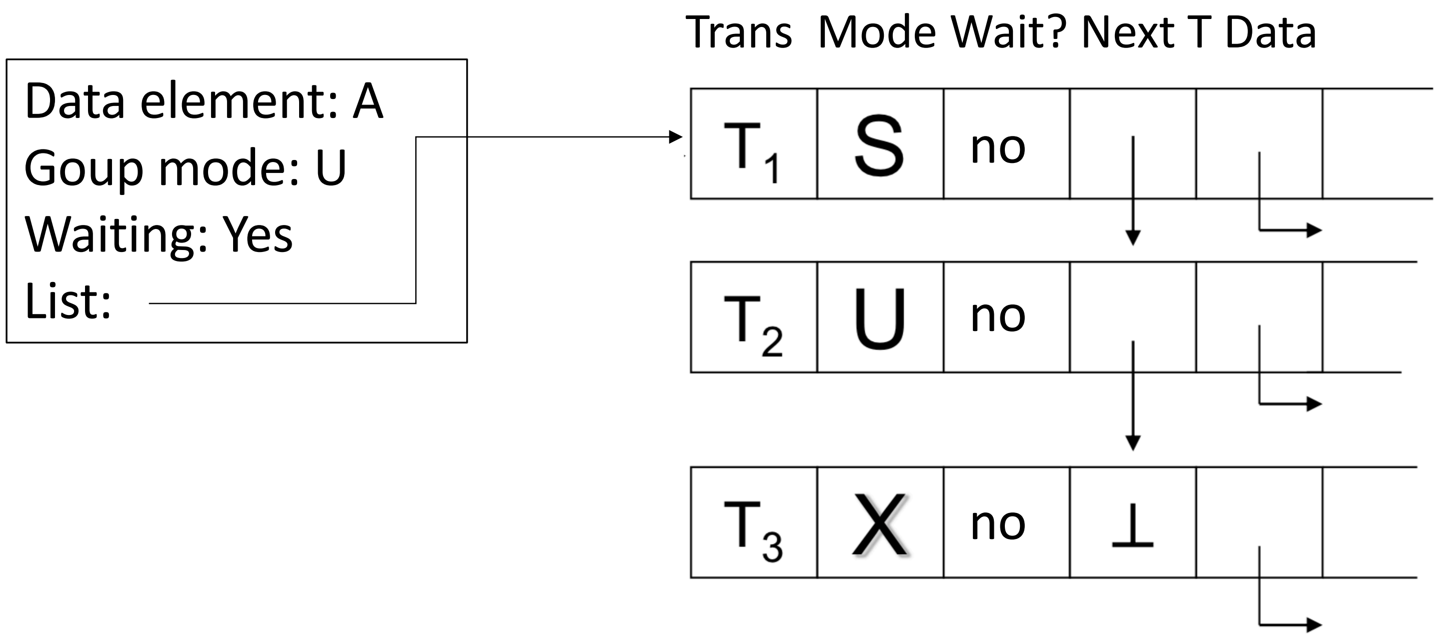
The lock manager consists of two parts:

- **Part I** analyzes each transaction T and inserts "correct" lock requirements prior to operations in T and sets the requirements in the lock table. The requirements it selects depend on which lock types are available.
- **Part II** controls whether the operations and lock requirements it receives from Part I can be performed. Those that cannot be realized are placed in a queue to wait for the lock that prevents execution to be removed (which also means that there is a queue for each lock)
- When T does commit (or abort), Part I deletes all locks set by T and notifies Part II, which checks the queues for these locks and allows the transactions that can continue

Lock table

- Logically, the lock table is a table that contains all lock information for each data item in the database
- In practice, the lock table is organized as a hash table with the address of the data element´s address as the key
- Unlocked data elements are not included in the lock table
- The lock table is therefore proportional to the number of requested and granted locks, and not to the number of data elements
- For each A in the lock table, the following information is stored:
  - Group mode (strictest lock held on A)
  - A waiting flag that indicates whether someone is waiting to lock A
  - A list of those T that are waiting for lock on A

Example of lock info for a data element A



To other data elements T_3 has (pending) lock on (useful for commit / abort)

Granularity and alert (warning) locks

- The concept of a data element is intentionally undefined. Three natural granularities on data elements are:
  - **a relation:** the naturally largest (lockable) data element
  - **a block:** a relation consists of one or more blocks
  - **a tuple:** a block can contain one or more tuples
- Different transactions may require locks at all these levels at the same time
- To achieve this, we introduce alert (warning) locks, **IS** and **IX**, which state that we intend to put a read or write lock further down in the hierarchy, respectively.

Alert (warning) locks (continued)

Requests lock ↓

Has lock ↓	IS	IX	S	X
IS	Yes	Yes	Yes	No
IX	Yes	Yes	No	No
S	Yes	No	Yes	No
X	No	No	No	No

Example: T wants to write tuple A in block B in relation R

- If R has neither S-lock nor X-lock, T sets IX-lock on R
- If T gets IX-lock on R, it checks if B has S or X lock. If B does not have those, T puts IX-lock on B
- If T gets IX-lock on B, it checks if A has any locks. If A does not have any, T puts X-lock on A and then can write A

Note that if a transaction T has a write lock on R, no one else can write a tuple in R until T clears the lock

Managing phantom tuples

Example:

- We shall sum a field for all the tuples in a relation R
- Before summing, we put a read-lock on all the R tuples to ensure a consistent answer
- During the addition operation, another transaction inserts a new tuple in R, which makes the sum become wrong
- This is possible because the tuple did not exist when we put our reading locks. Such a tuple is called a phantom tuple!
- The solution is to put an IS (intended Shared Lock) lock on the relation. Then, no one can enter any new tuples until the lock is deleted.