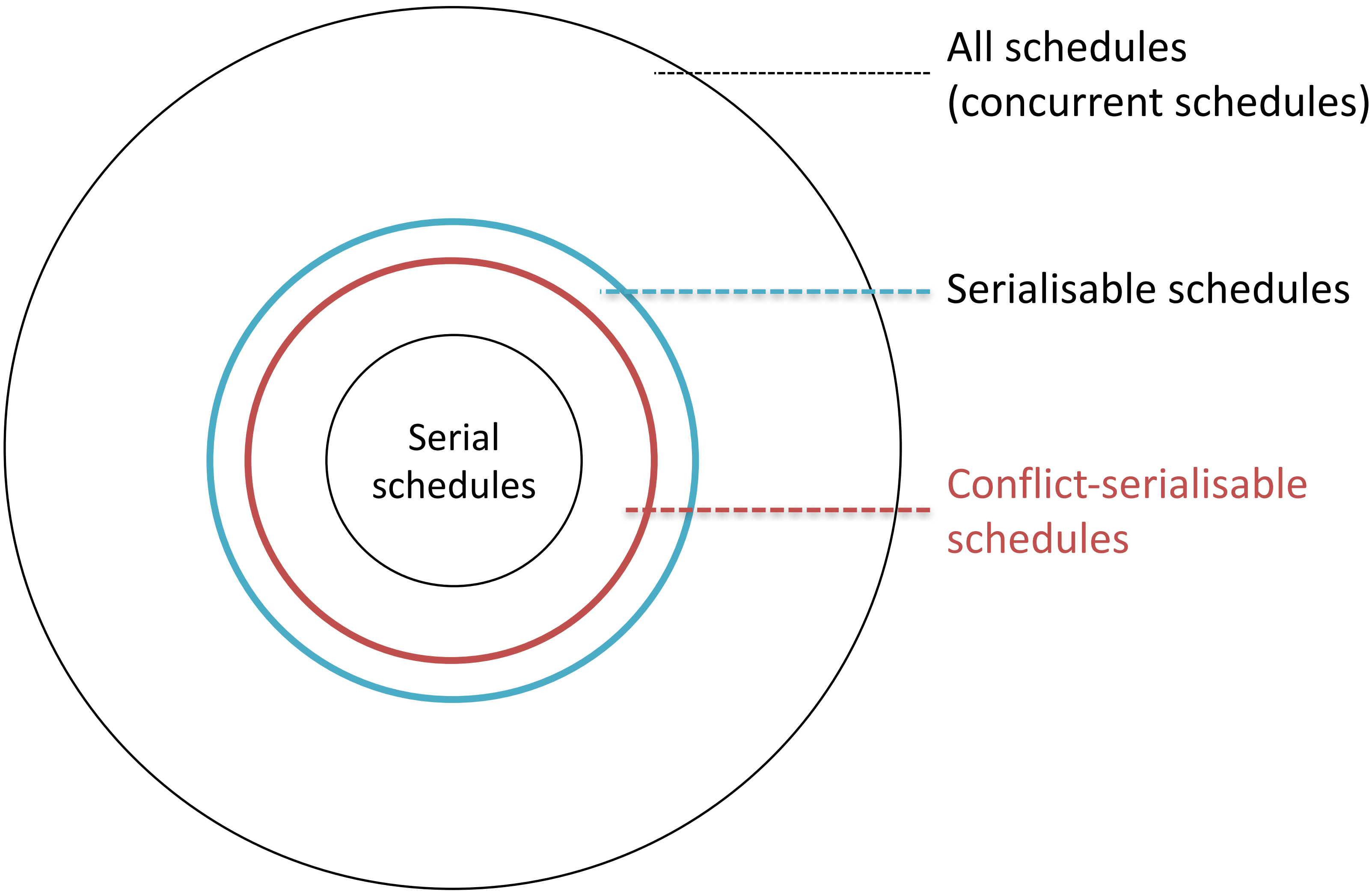
COMP207
Database Development

Lecture 6

Transaction Management:
Conflict-Serialisability using 2PL

Schedules Review



Schedule types

Schedule: Ordering of all operations in the transactions

- Serial schedule: First one entire transaction, then the next and so on
  - $r_1(x)$; $w_1(x)$; $r_1(y)$; $w_1(y)$; $r_2(x)$; $w_2(x)$;
- Concurrent schedule: Only requires operations in each transaction appears to appear in the same order
  - $r_1(x)$; $r_2(x)$; $w_1(x)$; $r_1(y)$; $w_2(x)$; c_2; $w_1(y)$; c_1
- Serialisable schedule: Effect on database is the same as some serial schedule
  - Hard to check

Conflict – Characterisation

from Lecture 5

- A **conflict** in a schedule is a pair of operations from different transactions such that:
  - the operations access the same item
  - at least one of them is a write operation
- Example:

$$S: r_1(X);\ \underline{w_1(X);}\ \underline{r_2(X);}\ w_2(X);\ r_1(Y);\ w_1(Y)$$

conflict in S

Conflict-Serialisability

from Lecture 5

- Two schedules S and S' are **conflict-equivalent** if S' can be obtained from S by swapping any number of *consecutive* non-conflicting operations from different transactions.

- Example:

S: $r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); w_1(Y)$ — Conflict-serialisable schedule

$r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y)$

$r_1(X); w_1(X); r_1(Y); r_2(X); w_2(X); w_1(Y)$

$r_1(X); w_1(X); r_1(Y); r_2(X); w_1(Y); w_2(X)$

S': $r_1(X); w_1(X); r_1(Y); w_1(Y); r_2(X); w_2(X)$ — Serial schedule

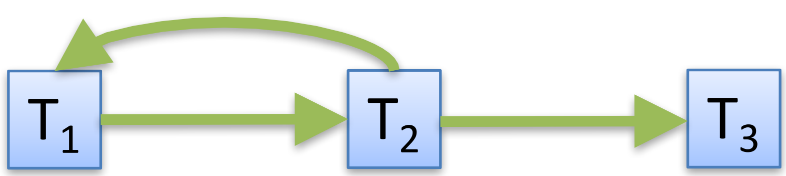
- A schedule is **conflict-serialisable** if it is conflict-equivalent to a serial schedule.

Precedence Graph

- The precedence graph for a schedule S is defined as follows:
  - It is a **directed graph.**
  - Its **nodes** are the transactions that occur in S.
  - It has an **edge** from transaction T_i to transaction T_j if there is a conflicting pair of operations op_1 and op_2 in S such that
    - op_1 appears before op_2 in S
    - op_1 belongs to transaction T_i
    - op_2 belongs to transaction T_j.
- Example:

S: $r_2(X); r_1(Y); w_2(X); r_2(Y); r_3(X); w_1(Y); w_3(X); w_2(Y)$

Precedence graph for S:



Testing Conflict-Serialisability

- To test if a schedule S is **conflict-serialisable**:
  - Construct the precedence graph for S.
  - If the precedence graph is **acyclic**, then S is conflict-serialisable. Otherwise not.

Acyclic graph: graph without a directed cycle

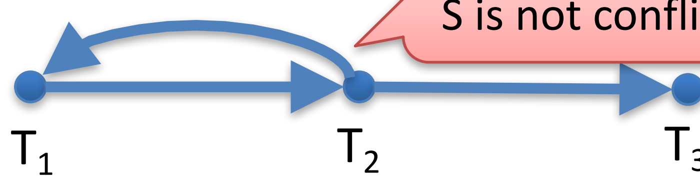
- Example 1: S: $r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); w_1(Y); r_2(Y); w_2(Y)$

  Precedence graph for S:

has no cycle → S is conflict-serialisable

- Example 2: S: $r_2(X); r_1(Y); w_2(X); r_2(Y); r_3(X); w_1(Y); w_3(X); w_2(Y)$

  Precedence graph for S:

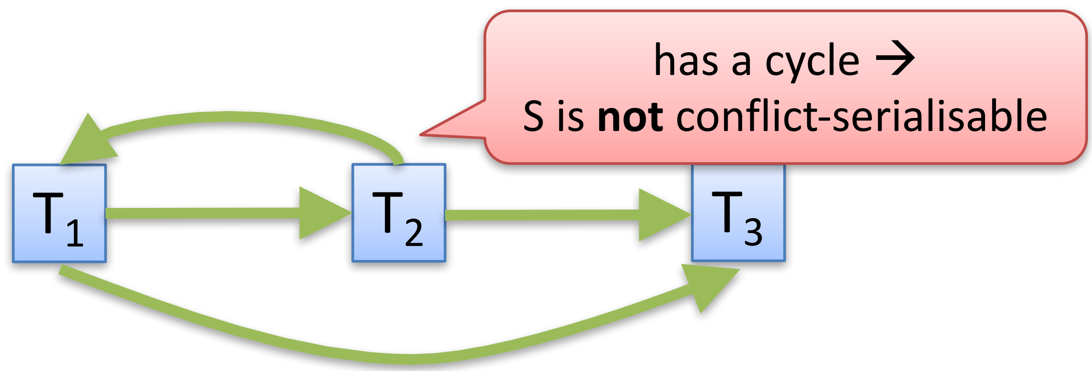


contains a cycle → S is not conflict-serialisable

Example 1

- Verify that the schedule below is **not** conflict-serialisable (using a precedence graph).

S: $w_2(X)$; $w_1(X)$; $w_1(Y)$; $w_2(Y)$; $w_3(X)$;



Example 2

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

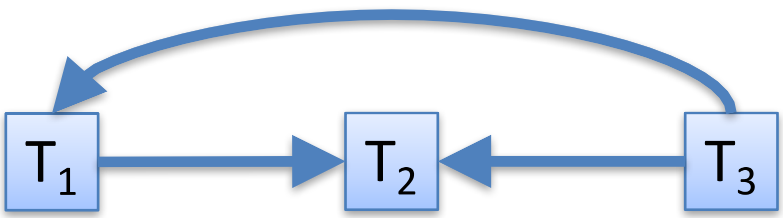
S: $r_1(Y), r_3(Y), r_1(X), r_2(X), w_2(X), r_3(Z), w_3(Z), r_1(Z), w_1(Y), r_2(Z)$

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

Example 2

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

S: $r_1(Y), r_3(Y), r_1(X), r_2(X), w_2(X), r_3(Z), w_3(Z), r_1(Z), w_1(Y), r_2(Z)$



has no cycle → S is conflict-serialisable

Any topological ordering yields a conflict-equivalent serial schedule

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

$r_3(Y), r_3(Z), w_3(Z)$

Example 2

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

S: $r_1(Y), r_3(Y), r_1(X), r_2(X), w_2(X), r_3(Z), w_3(Z), r_1(Z), w_1(Y), r_2(Z)$

has no cycle → S is conflict-serialisable

Any topological ordering yields a conflict-equivalent serial schedule

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

$r_3(Y), r_3(Z), w_3(Z)$

Example 2

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

S: $r_1(Y), r_3(Y), r_1(X), r_2(X), w_2(X), r_3(Z), w_3(Z), r_1(Z), w_1(Y), r_2(Z)$

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

$r_3(Y), r_3(Z), w_3(Z), r_1(Y), r_1(X), r_1(Z), w_1(Y),$

Example 2

- Verify that the schedule below is conflict-serialisable (using a precedence graph).

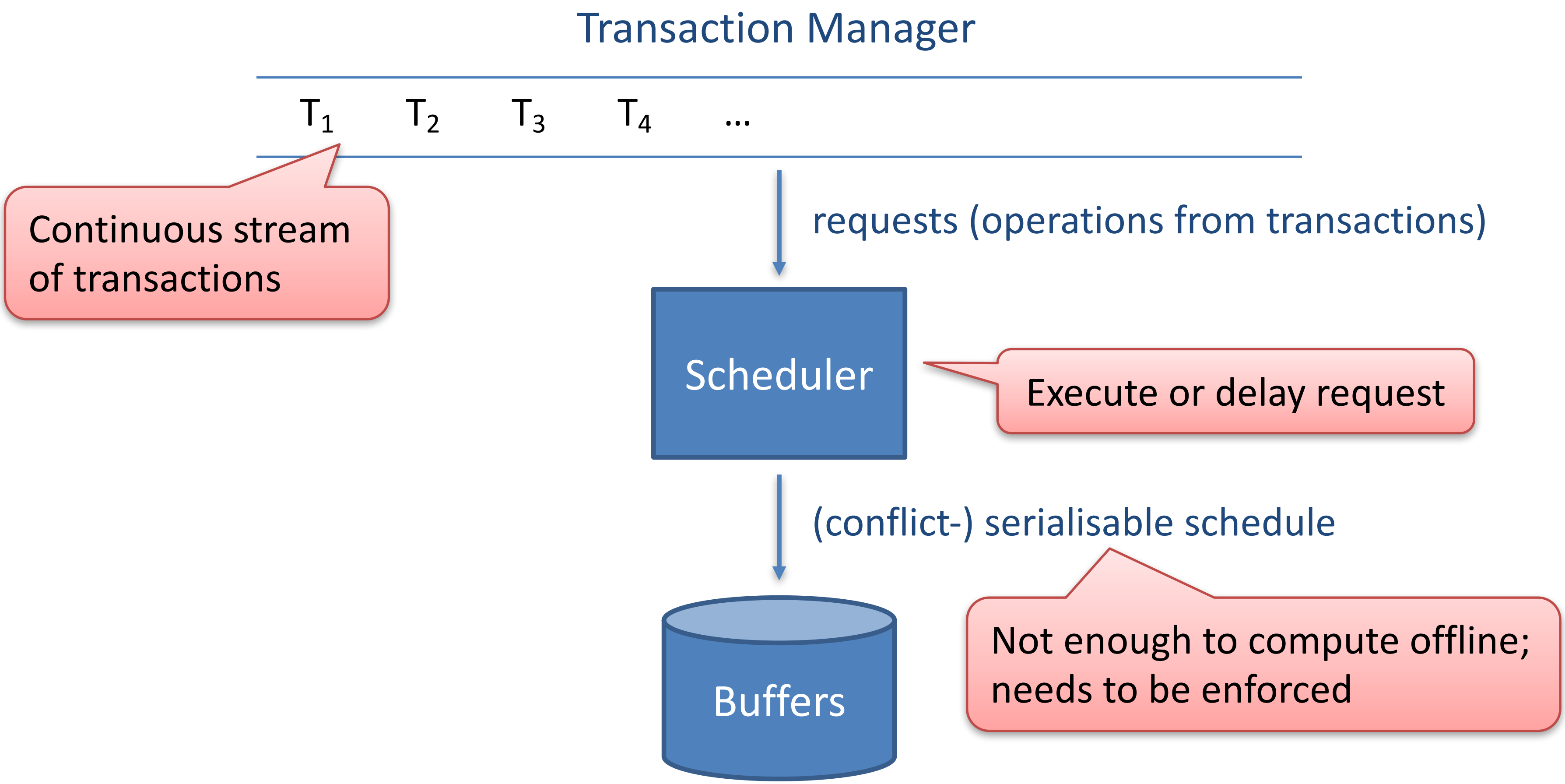
S: $r_1(Y)$, $r_3(Y)$, $r_1(X)$, $r_2(X)$, $w_2(X)$, $r_3(Z)$, $w_3(Z)$, $r_1(Z)$, $w_1(Y)$, $r_2(Z)$

- Can you find a conflict-equivalent serial schedule (using your precedence graph)?

$r_3(Y)$, $r_3(Z)$, $w_3(Z)$, $r_1(Y)$, $r_1(X)$, $r_1(Z)$, $w_1(Y)$, $r_2(X)$, $w_2(X)$, $r_2(Z)$

Are we done?

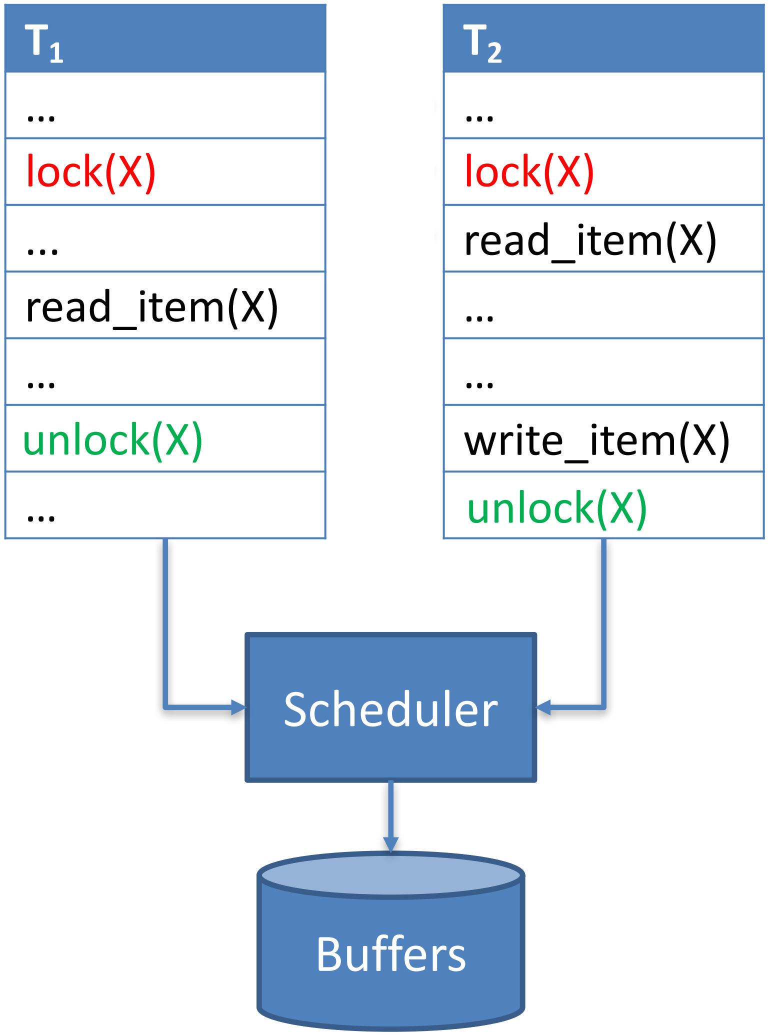
Transaction Scheduling in a DBMS



Enforcing Conflict-Serialisability Using Locks

Simple Locking Mechanism

- A transaction has to lock an item before it accesses it.
- Locks are requested from & granted by the scheduler:
  - Each item is locked by at most one transaction at a time.
  - Transactions wait until a lock can be granted.
- Each lock has to be released (unlocked) eventually.



Schedules With Simple Locks

- Extend syntax for schedules by two operations:
  - $l_i(X)$: transaction i requests a lock for item X
  - $u_i(X)$: transaction i unlocks item X
- Example:

  S: $l_1(X); r_1(X); u_1(X); l_2(X); r_2(X); w_2(X); u_2(X)$

- Rules:
  - For each $r_i(X)$ / $w_i(X)$ there is an earlier $l_i(X)$ without any $u_i(X)$ occurring between $l_i(X)$ and $r_i(X)$ / $w_i(X)$.
  - For each $l_i(X)$ there is a later $u_i(X)$.
  - If $l_i(X)$ comes before $l_j(X)$, then $u_i(X)$ occurs between $l_i(X)$ and $l_j(X)$.

… May Not Be Serialisable

T_1
lock(X)
read_item(X)
X := X + 100
write_item(X)
unlock(X)
lock(Y)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)

T_2
lock(X)
read_item(X)
X := 2*X
write_item(X)
unlock(X)
lock(Y)
read_item(Y)
Y := 2*Y
write_item(Y)
unlock(Y)

not serialisable (why?)

S: $l_1(X); r_1(X); w_1(X); u_1(X); l_2(X); r_2(X); w_2(X); u_2(X);$
$l_2(Y); r_2(Y); w_2(Y); u_2(Y); l_1(Y); r_1(Y); w_1(Y); u_1(Y)$

A Serialisable Schedule With Locks

T_1
lock(X)
read_item(X)
X := X + 100
write_item(X)
unlock(X)
lock(Y)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)

T_2
lock(X)
read_item(X)
X := 2*X
write_item(X)
unlock(X)
lock(Y)
read_item(Y)
Y := 2*Y
write_item(Y)
unlock(Y)

A Serialisable Schedule With Locks

T_1
lock(X)
read_item(X)
X := X + 100
write_item(X)
lock(Y)
unlock(X)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)

T_2
lock(X)
read_item(X)
X := 2*X
write_item(X)
lock(Y)
unlock(X)
read_item(Y)
Y := 2*Y
write_item(Y)
unlock(Y)

conflict-serialisable (why?)

T_2's request for lock on Y denied

S: $l_1(X)$; $r_1(X)$; $w_1(X)$; $l_1(Y)$; $u_1(X)$; $l_2(X)$; $r_2(X)$; $w_2(X)$; ____
$r_1(Y)$; $w_1(Y)$; $u_1(Y)$; $l_2(Y)$; $u_2(X)$; $r_2(Y)$; $w_2(Y)$; $u_2(Y)$

Two-Phase Locking (2PL)

- Simple modification of the simple locking mechanism that *guarantees conflict-serialisability*
- **Two-phase locking (2PL) condition:**
  In each transaction, all lock operations precede all unlocks.

“2PL transaction”

Transaction

Phase 1: request locks

+ possibly other
read/write operations

Phase 2: unlock

+ possibly other
read/write operations

Example 1

T_1
lock(X)
read_item(X)
X := X + 100
write_item(X)
unlock(X)
lock(Y)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)

T_2
lock(X)
read_item(X)
X := 2*X
write_item(X)
unlock(X)
lock(Y)
read_item(Y)
Y := 2*Y
write_item(Y)
unlock(Y)

2PL?

S: $l_1(X); r_1(X); w_1(X); u_1(X); l_2(X); r_2(X); w_2(X); u_2(X);$
$l_2(Y); r_2(Y); w_2(Y); u_2(Y); l_1(Y); r_1(Y); w_1(Y); u_1(Y)$

Example 2

T_1
lock(X)
read_item(X)
X := X + 100
write_item(X)
lock(Y)
unlock(X)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)

T_2
lock(X)
read_item(X)
X := 2*X
write_item(X)
lock(Y)
unlock(X)
read_item(Y)
Y := 2*Y
write_item(Y)
unlock(Y)

2PL?

S: $l_1(X);\ r_1(X);\ w_1(X);\ l_1(Y);\ u_1(X);\ l_2(X);\ r_2(X);\ w_2(X);$
$r_1(Y);\ w_1(Y);\ u_1(Y);\ l_2(Y);\ u_2(X);\ r_2(Y);\ w_2(Y);\ u_2(Y)$

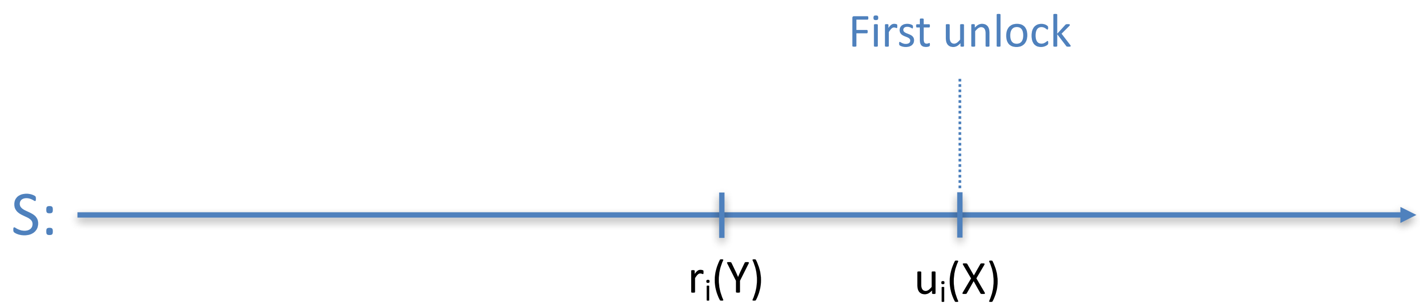
2PL Ensures Conflict-Serialisability

- If S is a schedule containing only 2PL transactions, then S is conflict-serialisable.
- Proof idea:

Claim: We can move all operations of T_i to beginning of schedule.

2PL Ensures Conflict-Serialisability

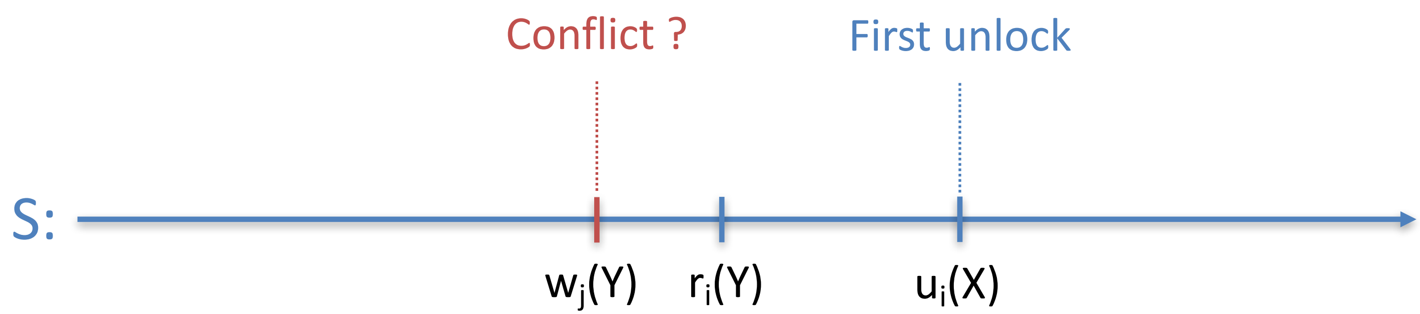
- If S is a schedule containing only 2PL transactions, then S is conflict-serialisable.
- Proof idea:



Claim: We can move all operations of T_i to beginning of schedule.

2PL Ensures Conflict-Serialisability

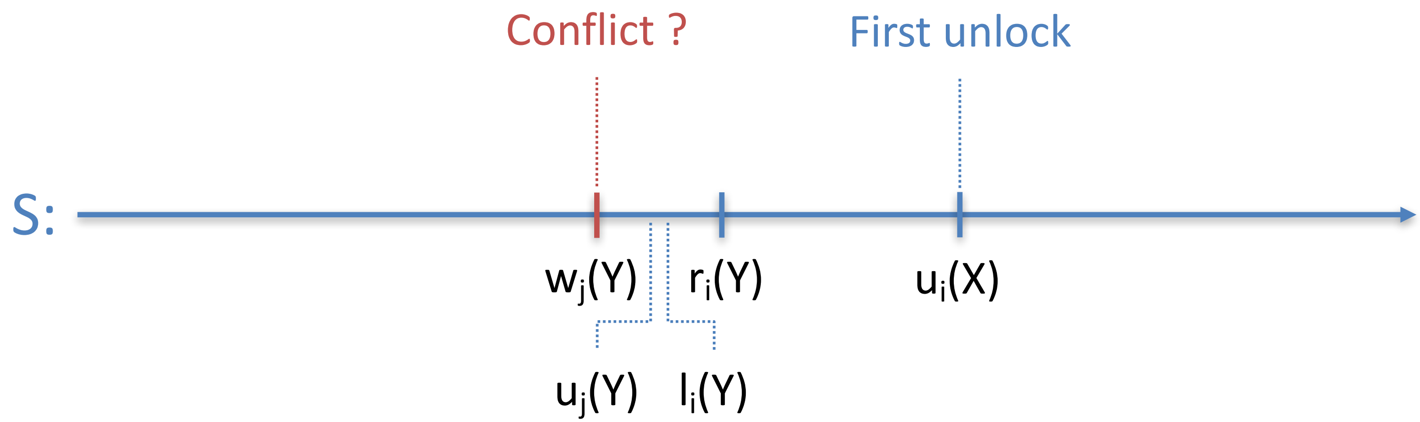
- If S is a schedule containing only 2PL transactions, then S is conflict-serialisable.
- Proof idea:



Claim: We can move all operations of T_i to beginning of schedule.

2PL Ensures Conflict-Serialisability

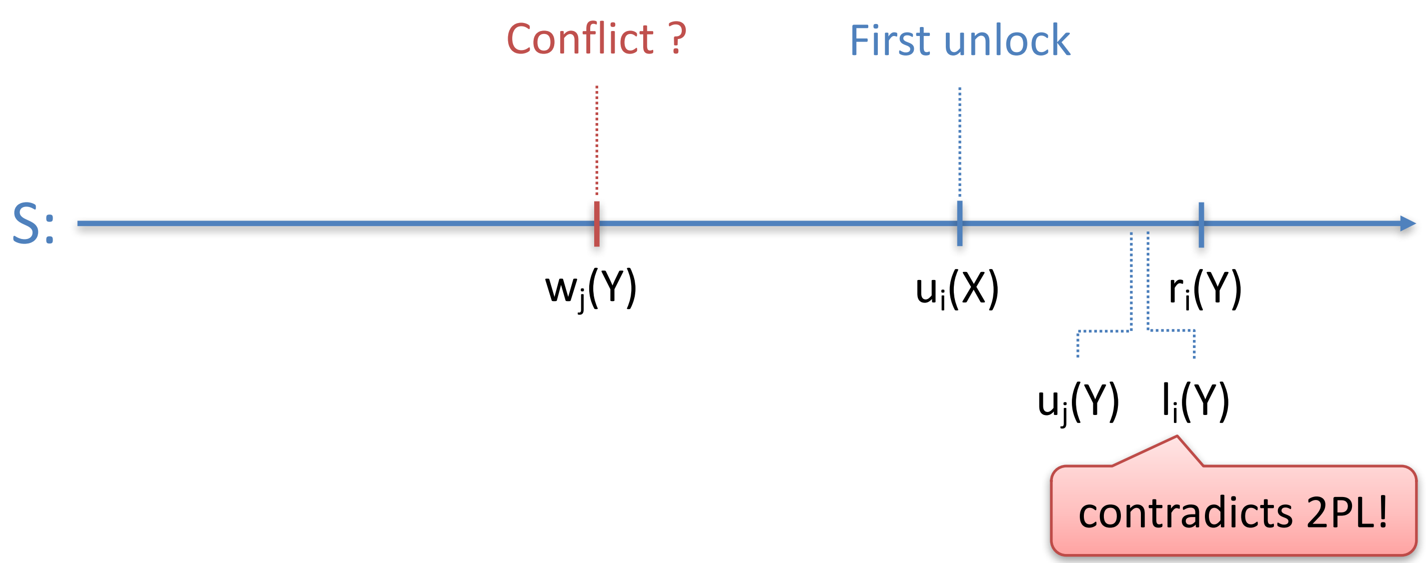
- If S is a schedule containing only 2PL transactions, then S is conflict-serialisable.
- Proof idea:



Claim: We can move all operations of T_i to beginning of schedule.

2PL Ensures Conflict-Serialisability

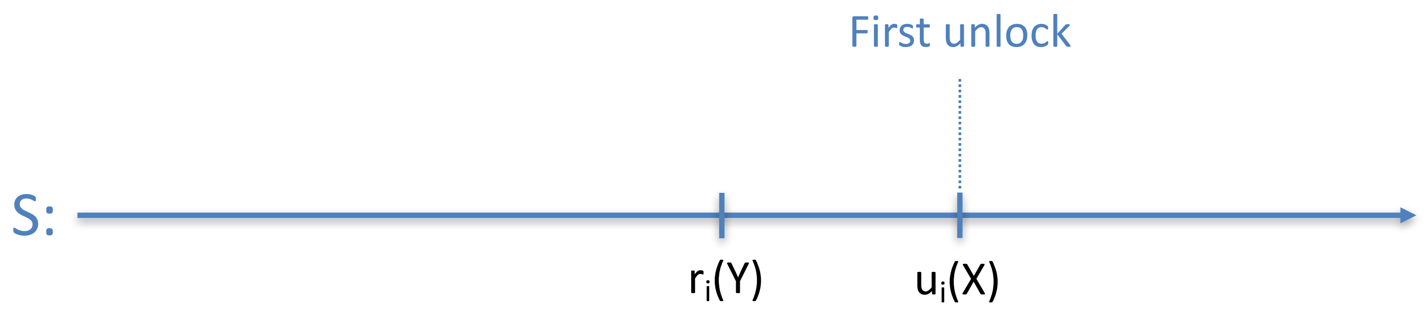
- If S is a schedule containing only 2PL transactions, then S is conflict-serialisable.
- Proof idea:



Claim: We can move all operations of T_i to beginning of schedule.

2PL Ensures Conflict-Serialisability

- If S is a schedule containing only 2PL transactions, then S is conflict-serialisable.
- Proof idea:



Claim: We can move all operations of T_i to beginning of schedule.

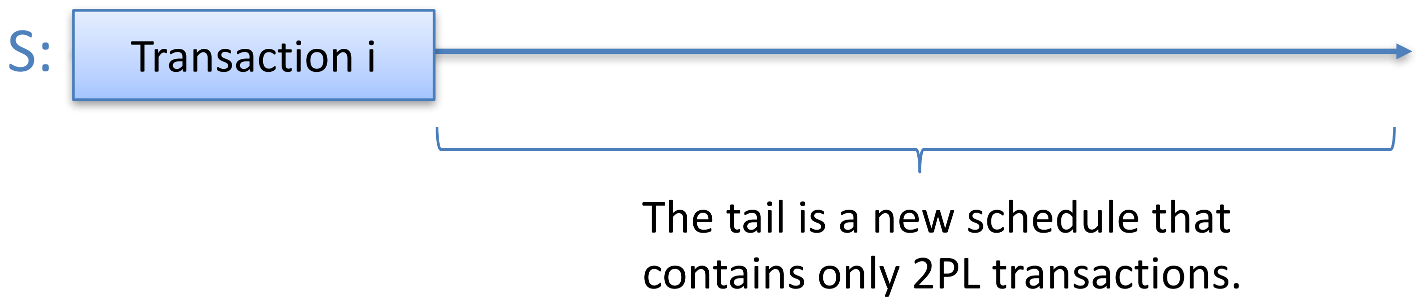
2PL Ensures Conflict-Serialisability

- If S is a schedule containing only 2PL transactions, then S is conflict-serialisable.
- Proof idea:

Claim: We can move all operations of T_i to beginning of schedule.

2PL Ensures Conflict-Serialisability

- If S is a schedule containing only 2PL transactions, then S is conflict-serialisable.
- Proof idea:



Repeat the procedure for the tail.

Still Some Issues

- 2PL ensures conflict-serialisability, but might lead to
  - **Deadlocks**: transactions might be forced to wait forever
  - Other issues (later)

Risk of Deadlocks

We will see later how to solve this problem.

T_1
lock(X)
read_item(X)
X := X + 100
write_item(X)
lock(Y)
unlock(X)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)

T_2
lock(Y)
read_item(Y)
Y := 2*Y
write_item(Y)
lock(X)
unlock(Y)
read_item(X)
X := 2*X
write_item(X)
unlock(X)

T_2's request for lock on X denied

$l_1(X); r_1(X); w_1(X);\ l_2(Y); r_2(Y); w_2(Y);$ _?_

T_1's request for lock on Y denied

Still Some Issues

- 2PL ensures conflict-serialisability, but might lead to
  - **Deadlocks**: transactions might be forced to wait forever
  - Other issues (later)
- **Overly simple locking mechanism**:
  - Have to lock an item X even if we only want to read it.
  - This delays all other transactions who want to access X, even if they only want to read X.
  - But: it would do no harm if several transactions read X (but don't write)

How can we make 2PL more flexible?

(e.g., allow read-only access by multiple transactions)

Solution: **different lock modes**

Shared & Exclusive Locks

- **Shared lock ("read lock"):**
  - Requested by transactions to read an item X
  - Granted to *several transactions at the same time*

Operation: **s-lock(X)**

- **Exclusive lock ("write lock"):**
  - Requested by transactions to write an item X
  - Granted to *at most one transaction at a time*

Operation: **x-lock(X)**

- Additional rules:
  - Shared lock on X is granted only if no *other* transaction holds an exclusive lock on X.
  - Exclusive lock on X is granted only if no *other* transaction holds a shared lock on X.

Schedules With Shared/Exclusive Locks

- Shorthand notation:
  - **$sl_i(X)$**: transaction i requests a *shared* lock for item X
  - **$xl_i(X)$**: transaction i requests an *exclusive* lock for item X
  - **$u_i(X)$**: transaction i releases all locks on item X
- Example:

T_1
s-lock(X)
read_item(X)
unlock(X)

T_2
s-lock(X)
read_item(X)
x-lock(X)
write_item(X)
unlock(X)

S: $sl_1(X)$; $r_1(X)$;
$sl_2(X)$; $r_2(X)$;
$u_1(X)$;
$xl_2(X)$; $w_2(X)$; $u_2(X)$

- Note: An individual transaction may hold both a shared lock and an exclusive lock for the same item X.

Problems With "Upgrading" Locks

- A shared lock on an item X can be upgraded later to an exclusive lock on X.
- Can use this to be "friendly" to other transactions.
- Caveat: risk of deadlock

T_1
s-lock(X)
read_item(X)
x-lock(X)
write_item(X)
unlock(X)

T_2
s-lock(X)
read_item(X)
x-lock(X)
write_item(X)
unlock(X)

$sl_1(X)$; $r_1(X)$; $sl_2(X)$; $r_2(X)$; ___?___

Update Locks to the Rescue

- **Update lock**:
  - Requested by transactions to read (not write) an item
  - May be upgraded later to an exclusive lock (shared locks can no longer be upgraded)
  - Granted to *at most one transaction at a time*
- New upgrading policy:

Operation:
u-lock(X)
or
$ul_i(X)$

Not symmetric

Transaction requests lock of type …

	Shared	**Update**	**Exclusive**
Shared	yes	yes	no
Update	no	no	no
Exclusive	no	no	no

Grant if the only types of locks held by *other* transactions are those with a “yes”

Example 1: Avoiding the Deadlock

No longer possible: Shared locks can no longer be upgraded. This now requires an update lock.

T_1
s-lock(X)
read_item(X)
x-lock(X)
write_item(X)
unlock(X)

T_2
s-lock(X)
read_item(X)
x-lock(X)
write_item(X)
unlock(X)

Example 1: Avoiding the Deadlock

T_1
u-lock(X)
read_item(X)
x-lock(X)
write_item(X)
unlock(X)

T_2
u-lock(X)
read_item(X)
x-lock(X)
write_item(X)
unlock(X)

$ul_1(X)$; $r_1(X)$; ____

T_2's request for update lock on X is denied

$xl_1(X)$; $w_1(X)$; $u_1(X)$;

$ul_2(X)$; $r_2(X)$; $xl_2(X)$; $w_2(X)$; $u_2(X)$

Example 2

T_1
s-lock(X)
read_item(X)
unlock(X)

T_2
u-lock(X)
read_item(X)
x-lock(X)
write_item(X)
unlock(X)

T_3
s-lock(X)
read_item(X)
unlock(X)

T_2 can request an update lock on X even though T_1 holds a shared lock on X

T_2's request for exclusive lock on X is denied (T_1 holds shared lock)

T_3's request for shared lock on X is denied (T_2 holds update lock)

$sl_1(X)$; $r_1(X)$; $ul_2(X)$; $r_2(X)$;

$u_1(X)$; $xl_2(X)$; $w_2(X)$; $u_2(X)$;

$sl_3(X)$; $r_3(X)$; $u_3(X)$

Two-Phase Locking (2PL) With Shared/Exclusive/Update Locks

- Straightforward generalisation:
  In each transaction, all lock operations (i.e., shared, exclusive, or update lock requests) precede all unlocks.

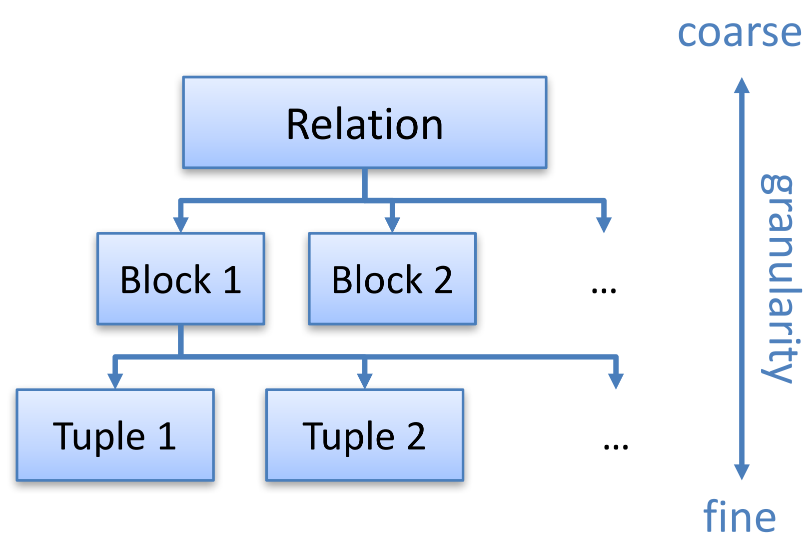
“2PL transaction”

Transaction
Phase 1: request locks + possibly other read/write operations
Phase 2: unlock + possibly other read/write operations

- Still guarantees conflict-serialisability.

Locks With Multiple Granularity

- DBMS may use locks at different levels of granularity
  - May lock relations
  - May lock disk blocks
  - May lock tuples



Shared lock on tuple suffices

- Examples:
  - SELECT name FROM Student WHERE studentID = 123456;
  - SELECT avg(salary) FROM Employee;

Shared lock on relation might be necessary

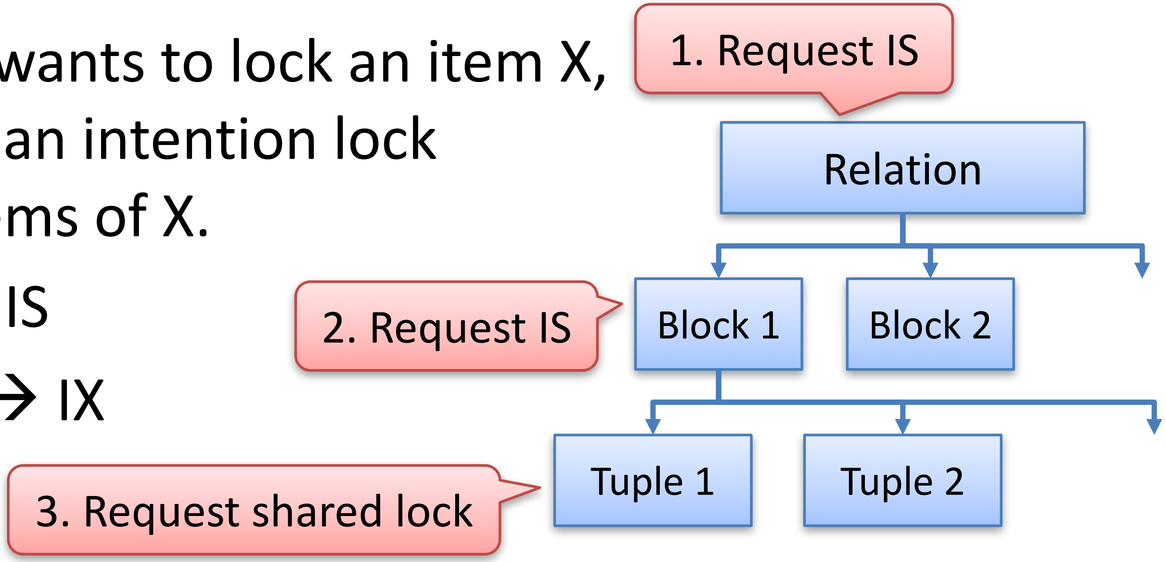
Trade-Offs

- Locking at **too coarse** granularity:
  - Low overhead (don’t need to store too much information)
  - Less degree of concurrency: may cause unnecessary delays
- Locking at **too fine** granularity:
  - High overhead: need to keep track of all locked items
  - High degree of concurrency: no unnecessary delays
- Need to prevent issues such as the following to guarantee (conflict-) serialisability:
  - A transactions holds shared lock for a tuple.
  - Another transaction holds exclusive lock for the relation.

Intention Locks

(a.k.a. Warning Locks)

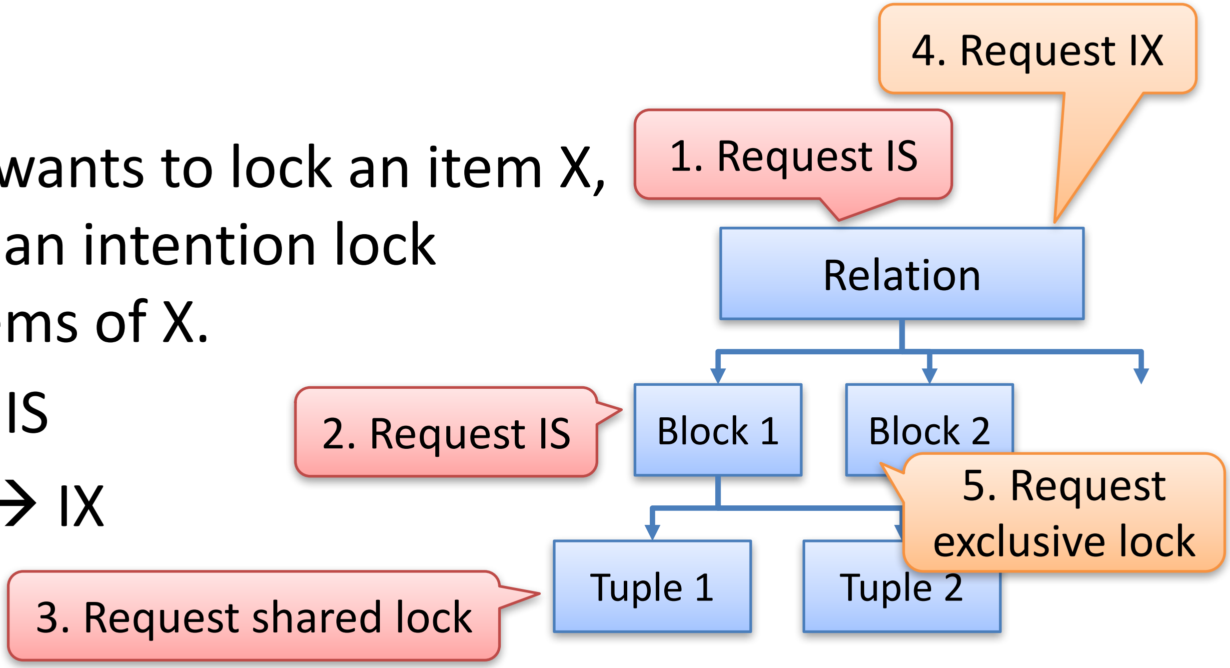
- We use shared and exclusive locks (no update locks)
- New **intention locks**:
  - **IS**: Intention to request a shared lock on a sub-item
  - **IX**: Intention to request an exclusive lock on a sub-item
- Rules:
  - If a transaction wants to lock an item X, it must *first* put an intention lock on the super-items of X.
  - Shared locks → IS
  - Exclusive locks → IX



Intention Locks

(a.k.a. Warning Locks)

- We use shared and exclusive locks (no update locks)
- New **intention locks**:
  - **IS**: Intention to request a shared lock on a sub-item
  - **IX**: Intention to request an exclusive lock on a sub-item
- Rules:
  - If a transaction wants to lock an item X, it must *first* put an intention lock on the super-items of X.
  - Shared locks → IS
  - Exclusive locks → IX



Policy for Granting Locks

Transaction requests lock of type …

	Shared (S)	Exclusive (X)	IS	IX
Shared (S)	yes	no	yes	no
Exclusive (X)	no	no	no	no
IS	yes	no	yes	yes
IX	no	no	yes	yes

Grant if the only types of locks held by *other* transactions are those with a "yes"

Summary

- How to test & enforce conflict-serialisability
- Testing: via preference graphs
  - Easy to construct the graph
  - Then: simple test for acyclicity
- Enforcing: e.g., via locking
  - 2PL ensures conflict-serialisability
  - Some types of locks