Fundamentals of Database Systems

Transaction Management

Jef Wijsen

University of Mons-Hainaut (UMH)

Transaction Management

PART II:

Concurrency Control

TOC Concurrency Control

- Serializable schedules
  - Correct and incorrect interleavings
  - Serial and serializable schedules
  - Testing serializability
- Two-Phase Locking (2PL)
- Concurrency control by timestamps
- Isolation levels in SQL2
- Exercises

Serializable schedules

Correct and incorrect interleavings

Recall ACID

- A transaction transforms a consistent database state into a new consistent database state (C of ACID).
- In general, a transaction will execute multiple writes; the database will generally be inconsistent in between two such writes.
- However, such inconsistent intermediate states should be hidden to other concurrent transactions (I of ACID).
- The overall effect should be as if each trx had executed in its entirety at a single time instant.

Two Trx (1)

T
READ(A,v)
READ(B,w)
v:=v+w
WRITE(A,v)
WRITE(B,0)

U
READ(B,u)
u:=2*u
WRITE(B,u)

Two Trx (2)

- Think of A and B as number of Euros owned by An and Bob respectively.
  Assume initially A=a and B=b (a,b ≥ 0).
- Trx T gives Bob's money to An.
  Trx U doubles Bob's wealth.
- Executing both trx once should yield either
  - **A=a+b, B=0 (T followed by U, denoted [T,U]), or**
  - **A=a+2b, B=0 (U followed by T, denoted [U,T]).**

  Either state is consistent!

Interleaving I: Incorrect

T	U	u	v	w	A	B
	READ(B,u)	b			a	b
	u:=2*u	2b			a	b
READ(A,v)		2b	a		a	b
READ(B,w)		[illegible]	[illegible]	[illegible]	[illegible]	b
v:=v+w		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
WRITE(A,v)		[illegible]	[illegible]	[illegible]	[illegible]	b
WRITE(B,0)		2b	a+b	b	a+b	0
	WRITE(B,u)	2b	a+b	b	a+b	2b

The problem is that T and U concurrently update the same old value B=b.

Result generally inconsistent (consistent if b=0)

Interleaving II: Same Effect as [U,T] (and hence correct)

T	U	u	v	w	A	B
	READ(B,u)	b			a	b
READ(A,v)		b	a		a	b
	u:=2*u	2b	a		a	b
	WRITE(B,u)	2b	a		a	2b
READ(B,w)		2b	a	2b	a	2b
v:=v+w		[illegible]	a+2b	2b	a	2b
WRITE(A,v)		[illegible]	a+2b	2b	a+2b	2b
WRITE(B,0)		[illegible]	a+2b	2b	a+2b	0

T reads the new value B=2b written by U.

Result consistent: same effect as [U,T]

Interleaving III: Correct by Coincidence

T	U	u	v	w	A	B
	READ(B,u)	b			a	b
	u:=2*u	2b			a	b
READ(A,v)		2b	[illegible]	[illegible]	[illegible]	b
READ(B,w)		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
v:=v+w		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
WRITE(A,v)		[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	WRITE(B,u)	2b	a+b	b	a+b	2b
WRITE(B,0)		2b	a+b	b	a+b	0

Note that T and U concurrently update the same old value B=b.

Result consistent by coincidence

…

Arithmetic Coincidence

- The arithmetic coincidence in ‘Interleaving III’ is that 0=2*0. That is, we can always interprete B=0 as having Bob’s wealth doubled after executing T.
- The coincidence would not occur if we had, for example, ‘u:=2+u’ instead of ‘u:=2*u’ in U.

  Interleaving III would still yield A=a+b, B=0, but the only consistent outcomes would be:

  - **A=a+b, B=2 (T followed by U, denoted [T,U]), or**
  - **A=a+b+2, B=0 (U followed by T, denoted [U,T]).**
- Note that ‘Interleaving II,’ being equivalent to [U,T], would necessarily yield A=a+b+2, B=0.

Aim of Concurrency Control

- Characterize correct interleavings.
- Since recognizing arithmetic coincidences is generally impossible, we focus on interleavings whose correctness relies solely on the ordering of READ and WRITE operations.
- Next, we investigate how correctness of interleavings can be ensured in an operational system.

Serializable schedules

Serial and serializable schedules

Shedules (1)

- We usually name our trx T_1, T_2, T_3,…
  In particular, let the example trx T and U be denoted by T_1 and T_2 in what follows.
- If we accept that only READ and WRITE operations matter, T_1 (formerly T) can be expressed as '$R_1(A)R_1(B)W_1(A)W_1(B)$' and T_2 (formerly U) as '$R_2(B)W_2(B)$', where R and W indicate reads and writes resp.
- Interleaving II is expressed as: '$R_2(B)R_1(A)W_2(B)R_1(B)W_1(A)W_1(B)$'.
- Such interleaving is called a schedule.

Schedules (2)

- So a schedule of trx $T_1,T_2,\ldots,T_n$ is a sequence S of the actions occurring in $T_1,T_2,\ldots,T_n$ such that the actions of each T_i appear in S in the same order that they appear in T_i $(1 \leq i \leq n)$.
- Actions of trx T_i are either $R_i(.)$ or $W_i(.)$.
- A schedule is serial if no two actions of the same trx are separated by an action of a different trx.
- Examples of serial schedules are:
  - '$R_1(A)R_1(B)W_1(A)W_1(B)R_2(B)W_2(B)$' abbreviated as $[T_1,T_2]$.
  - '$R_2(B)W_2(B)R_1(A)R_1(B)W_1(A)W_1(B)$' abbreviated as $[T_2,T_1]$.

Why Must Interleaving II Be Correct?

- Since we accept consistency of trx (C of ACID), we must also accept that <u>serial schedules are correct</u>.
- Now recall Interleaving II:
  '$R_2(B)$ $\boxed{R_1(A) W_2(B)}$ $R_1(B) W_1(A) W_1(B)$'
  This is almost the serial schedule:
  '$R_2(B)$ $\boxed{W_2(B) R_1(A)}$ $R_1(B) W_1(A) W_1(B)$'
- Interleaving II is equal to the serial schedule up to a swapping of $R_1(A)$ and $W_2(B)$.
- Would such swap change the effect on the database? Evidently the answer is 'no.'

Conflicting Actions

- We say that two actions A and B of <u>different trx</u> **conflict** if the effect of AB can possibly be different from BA.
- Clearly, if A and B read or write <u>different database elements</u>, then they do not conflict. That is,

  if A is $R_i(X)$ or $W_i(X)$,
  B is $R_k(Y)$ or $W_k(Y)$,
  $X \neq Y$, and $i \neq k$,
  then A and B do not conflict.
- Two reads, even of the same element, never conflict.
- On the other hand (assuming $i \neq k$),
  - $R_i(X)$ and $W_k(X)$ conflict, for the value read by trx T_i is likely to differ in $R_i(X)W_k(X)$ and $W_k(X)R_i(X)$.
  - $W_i(X)$ and $W_k(X)$ conflict, for the final value written is likely to differ in $W_i(X)W_k(X)$ and $W_k(X)W_i(X)$.

Serializable Schedule

- A schedule is serializable if it can be turned into a serial schedule by repeatedly swapping neighboring non-conflicting actions of different trx.
- Never swap actions of the same trx!
- Since we accept that serial schedules are correct, we must also accept that serializable schedules are correct.
- Our approach to concurrency will be to require that schedules be serializable.
- Obviously, serial schedules are serializable.
- Note: the above notion is also called conflict-serializable in the literature.

Serializable schedules

Testing serializability

The ‘Game’ of Serializing

Initial schedule

$R_1(A)$	$W_1(A)$	$R_2(A)$	$W_2(A)$	$R_1(B)$	$W_1(B)$	$R_2(B)$	$W_2(B)$
$R_1(A)$	$W_1(A)$	$R_2(A)$	$R_1(B)$	$W_2(A)$	$W_1(B)$	$R_2(B)$	$W_2(B)$
$R_1(A)$	$W_1(A)$	$R_1(B)$	$R_2(A)$	$W_2(A)$	$W_1(B)$	$R_2(B)$	$W_2(B)$
$R_1(A)$	$W_1(A)$	$R_1(B)$	$R_2(A)$	$W_1(B)$	$W_2(A)$	$R_2(B)$	$W_2(B)$
$R_1(A)$	$W_1(A)$	$R_1(B)$	$W_1(B)$	$R_2(A)$	$W_2(A)$	$R_2(B)$	$W_2(B)$

Serial schedule with the same effect

Testing Serializability: Intuition

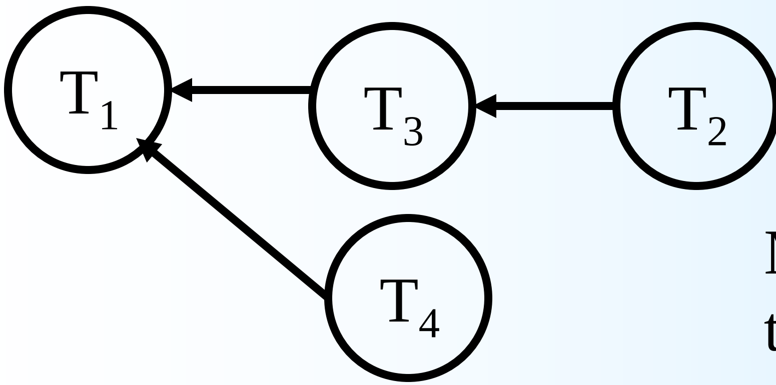
- Can we efficiently decide whether the ‘game’ can possibly reach a serial schedule?
- We can easily see that the schedule
  …$R_1(A)$…$W_2(A)$…$W_1(A)$ …
  is not serializable.
- Since $W_2(A)$ and $W_1(A)$ conflict, we cannot attain a serial schedule where T_1 precedes T_2. We write $T_2<T_1$ to denote that T_2 must precede T_1.
- Likewise, since $R_1(A)$ and $W_2(A)$ conflict, we have $T_1<T_2$.
- Since $T_2<T_1$ and $T_1<T_2$ are contradictory, we conclude that the schedule is not serializable.

Testing Serializability: Algorithm

- Let a schedule S be given.
- Construct a precedence graph as follows:
- Introduce a node labeled T_i for every trx T_i in S.
- Introduce an edge from node T_i to node T_k with $i \neq k$ if some action A of T_i is followed in S by some conflicting action B of T_k.
- Note that A and B must not be adjacent in S.
  An edge from T_i to node T_k agrees with $T_i < T_k$.
- S is serializable iff its precedence graph is acyclic.

Testing Serializability: Example 1

$R_4(C)W_3(A)R_1(A)W_1(C)R_2(B)W_3(B)$



Acyclic, hence serializable.

Moreover, the graph tells us that we can serialize into any one of:

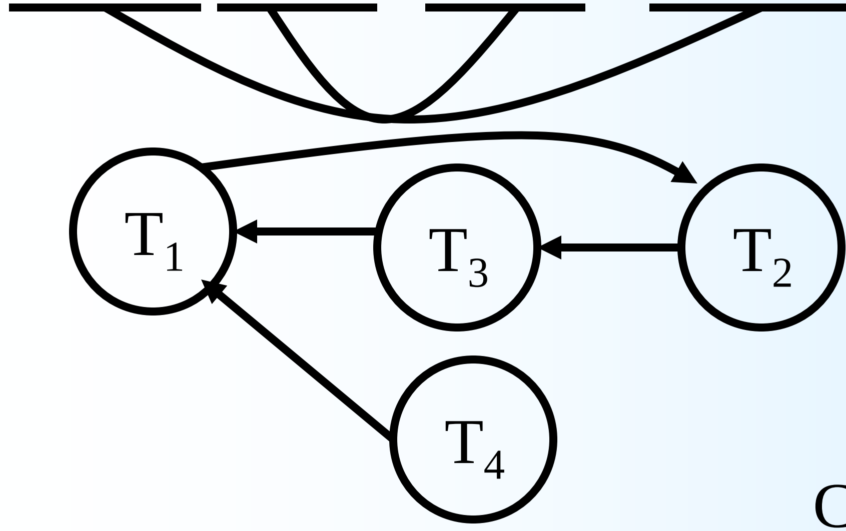
$\mathbf{T_2T_3T_4T_1}$

$\mathbf{T_2T_4T_3T_1}$

$\mathbf{T_4T_2T_3T_1}$

Testing Serializability: Example 2

$R_4(C)W_3(A)R_1(A)W_1(C)W_1(D)R_2(D)R_2(B)W_3(B)$



Cyclic, hence not serializable.

The Price to Pay for 'Simplicity'…

- By requiring serializability, we refuse some correct schedules…
- The schedule $W_1(A)W_2(A)W_1(A)$ is non-serializable because it is non-serial and all neighboring actions conflict. So it will be out of our scope.
- Nevertheless, as T_1 overwrites the value written by T_2 without ever reading it, the effect is that of $W_2(A)W_1(A)W_1(A)$.

TOC Concurrency Control

- Serializable schedules
- Two-Phase Locking (2PL)
  - The three rules of the protocol
  - Correctness proof
  - The locking scheduler
  - Multiple-granularity locking
  - Deadlock
  - Strict 2PL
- Concurrency control by timestamps
- Isolation levels in SQL2
- Exercises

Two-Phase Locking (2PL)

The three rules of the protocol

Ensuring Serializability

- The following approach is impractical/unfeasible:

  Execute trx in an <u>unconstrained manner</u>, periodically test for serializability, and break cycles by undoing trx…
- Unfeasible, because committed trx cannot be undone (remember D of ACID).
- Rather we will impose a <u>protocol</u>, called Two-Phase Locking (2PL), that guarantees that a schedule will be serializable.

Shared and Exclusive Locks

- A shared lock (S-lock) on a db element Y is a permission to read Y.
- An exclusive lock (X-lock) on Y is a permission to read or write Y.
- Operations:

$S_i(Y)$	T_i asks an S-lock on Y.
$X_i(Y)$	T_i asks an X-lock on Y.
$U_i(Y)$	T_i releases any lock it currently holds on Y (Unlock).

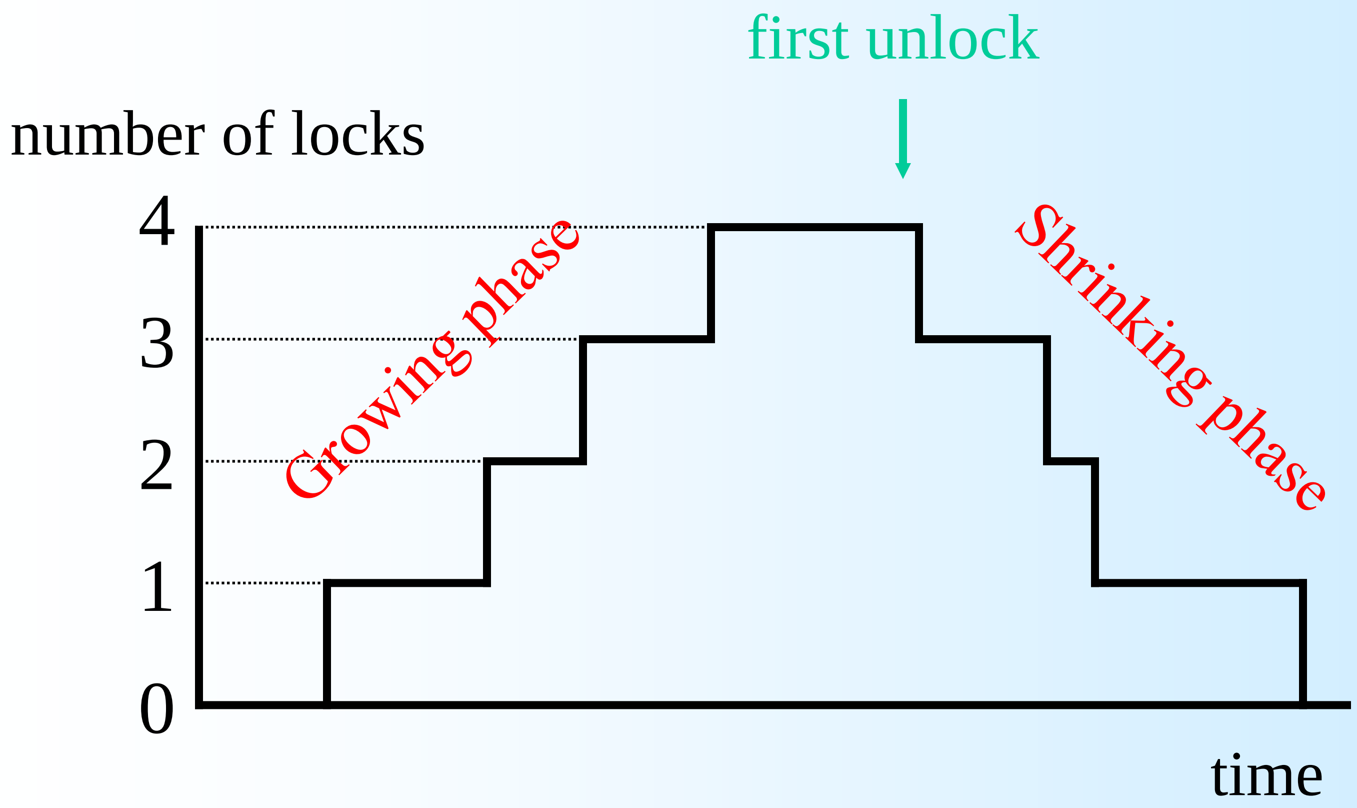
The Protocol 2PL (1)

- *Rule L1:* A trx must not read Y without holding an S-lock or an X-lock on Y. A trx must not write Y without holding an X-lock on Y. More precisely,
  - A read action $R_i(Y)$ must be preceded by $S_i(Y)$ or $X_i(Y)$, with no intervening $U_i(Y)$.
  - A write action $W_i(Y)$ must be preceded by $X_i(Y)$, with no intervening $U_i(Y)$.
  - All lock requests must be followed by an unlock of the same element.
- *Rule L2:* In every trx, all lock requests must precede all unlock requests.

Example of Rules *L1* and *L2*

- $R_1(A)W_1(B)$ could be extended as:
  $S_1(A)X_1(B)R_1(A)W_1(B)U_1(A)U_1(B)$,
  or as:
  $S_1(A)R_1(A)X_1(B)U_1(A)W_1(B)U_1(B)$.
- On the other hand,
  $S_1(A)R_1(A)\boxed{U_1(A)X_1(B)}W_1(B)U_1(B)$
  violates rule *L2* as the unlock request $U_1(A)$ precedes the lock request $X_1(B)$.

What’s in a Name?



The Protocol 2PL (2)

- *Rule L3:* Two trx cannot simultaneously hold a lock for conflicting actions. That is,
  - o $S_i(Y)$ and a following $X_k(Y)$ with $i \neq k$ must be separated by an intervening $U_i(Y)$.
  - o $X_i(Y)$ and a following $S_k(Y)$ or $X_k(Y)$ with $i \neq k$ must be separated by an intervening $U_i(Y)$.
- Schedules obeying all three rules are called 2PL-schedules.

2PL Summary

- Rule *L1:*
  - A shared or exclusive lock is needed for reading.
  - An exclusive lock is needed for writing.
  - All requested locks need to be released later on.
- Rule *L2:* Once you have released a lock, you are not allowed to ask any further lock later on.
- Rule *L3:* If some trx holds an exclusive lock on a db element, then no other trx can hold a shared or exclusive lock on that same element.

Example 2PL-Schedule

Turn $R_1(A)W_2(A)R_1(B)$ into a 2PL-schedule by adding lock and unlock requests, while preserving the order of reads and writes.

first unlock of T_1

$S_1(A)R_1(A)S_1(B)U_1(A)X_2(A)W_2(A)U_2(A)R_1(B)U_1(B)$

T_1 holds S-lock on A

T_2 holds X-lock on A

T_1 holds S-lock on B

Compatibility Matrix

The order of lock/unlock requests implied by rule *L3* can be summarized in a compatibility matrix:

Locks requested by some trx T

Locks held by some different trx U

	S	X
S	Yes	No
X	No	No

Lock Upgrade

- 2PL does not prevent a trx T_i from asking $X_i(Y)$ while holding an S-lock on Y.
  Such $X_i(Y)$ request is called a lock upgrade.
- For example, $R_1(A)W_1(A)$ can be turned into

  $X_1(A)R_1(A)W_1(A)U_1(A)$,

  but also into

  $S_1(A)R_1(A)X_1(A)W_1(A)U_1(A)$.

Two-Phase Locking (2PL)

Correctness proof

2PL Ensures Serializability

- Lemma. If an action of T_i is followed by a conflicting action of T_k ($i \neq k$) in a 2PL-schedule, then the first unlock of T_i precedes the first unlock of T_k.
- Theorem. Each 2PL-schedule can be serialized into a serial schedule where the trx appear in the order that they issue their first unlock.
- Corollary. Each 2PL-schedule is serializable.

Proof of Lemma (Sketch)

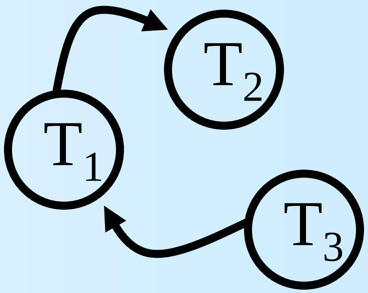
- Proof for write-only trx; generalization is easy.
- Assume $W_i(A)$ is followed by $W_k(A)$ in a 2PL-schedule.
- By rule *L1*, $W_i(A)$ must be preceded by $X_i(A)$, and $W_k(A)$ must be preceded by $X_k(A)$.
- By rule *L3*, $X_i(A)$ and $X_k(A)$ must be separated by $U_i(A)$, i.e., the schedule contains $U_i(A) \ldots X_k(A)$.
- No unlock of T_k can precede $U_i(A)$, or else T_k would violate rule *L2*.
- Hence, the first first unlock of T_i precedes the first unlock of T_k.

Proof of Theorem (Sketch)

- For a schedule S with two write-only trx T_1 and T_2.
- Assume without loss of generality that the first unlock of T_1 precedes the first unlock of T_2. We need to show that S can be serialized into [T_1,T_2].
- S cannot contain $W_2(Y) \ldots W_1(Y)$, or else, by the preceding lemma, the first unlock of T_2 precedes the first unlock of T_1, a contradiction.
- It follows that S can be serialized into [T_1,T_2].

The Price to Pay For 'Simplicity'…

$W_1(A)R_2(A)R_3(B)W_1(B)$



The precedence graph is acyclic,
so the schedule is serializable.
Can it be turned into a 2PL-schedule?

- By rules *L1* and *L3*, T_1 must issue $U_1(A)$ prior to $R_2(A)$.
- Because of $R_3(B)W_1(B)$, the first (and only) unlock $U_3(B)$ of T_3 must precede the first unlock of T_1 (cf. lemma).
- It follows that $U_3(B)$ must precede $R_2(A)$.
- But then T_2 cannot satisfy rules *L1* and *L2*…
- To conclude, in 2PL, the reads and writes cannot occur in exactly the order shown.

Two-Phase Locking (2PL)

The locking scheduler

Locking Scheduler

- Rules *L1* and *L2* are the responsibility of the trx in general (cf. discussion later).
- **Enforcing rule *L3* is the responsibility of a DBMS module, called the locking scheduler.**
- If a lock request by trx T is incompatible with a lock currently held by some other trx U, then T will be suspended and cannot be resumed before U has released its lock.

Lock Table

- The lock manager stores housekeeping information in a lock table.
- For example,

T_1 and T_2 hold
a shared lock an A

db element	locks held	wait queue
A	$\{S_1,S_2\}$	$\langle X_4,X_5\rangle$
B	$\{X_3\}$	$\langle S_1\rangle$

T_4 and T_5 are waiting for
an exclusive lock on A

Handling $S_1(A)$ Request

$$A \mid \{\} \mid \langle\rangle \longrightarrow A \mid \{S_1\} \mid \langle\rangle$$

$$A \mid \{S_2,S_3,\ldots\} \mid \langle\rangle \longrightarrow A \mid \{S_1,S_2,S_3,\ldots\} \mid \langle\rangle$$

$$A \mid \{S_2,S_3,\ldots\} \mid \langle X_4\rangle \xrightarrow{???} A \mid \{S_1,S_2,S_3,\ldots\} \mid \langle X_4\rangle$$

$$A \mid \{S_2,S_3,\ldots\} \mid \langle X_4\rangle \longrightarrow A \mid \{S_2,S_3,\ldots\} \mid \langle X_4,S_1\rangle$$

First-Come-First-Served
prevents starvation

$$A \mid \{X_2\} \mid \langle S_3,X_4\rangle \longrightarrow A \mid \{X_2\} \mid \langle S_3,X_4,S_1\rangle$$

Handling $X_1(A)$ Request

A | $\{\}$ | $\langle\rangle$ $\longrightarrow$	A | $\{X_1\}$ | $\langle\rangle$
A | $\{S_2,S_3,\ldots\}$ | $\langle\rangle$ $\longrightarrow$	A | $\{S_2,S_3,\ldots\}$ | $\langle X_1\rangle$
A | $\{S_2,S_3,\ldots\}$ | $\langle X_4\rangle$ $\longrightarrow$	A | $\{S_2,S_3,\ldots\}$ | $\langle X_4,X_1\rangle$
A | $\{X_2\}$ | $\langle\rangle$ $\longrightarrow$	A | $\{X_2\}$ | $\langle X_1\rangle$
A | $\{X_2\}$ | $\langle S_3,X_4\rangle$ $\longrightarrow$	A | $\{X_2\}$ | $\langle S_3,X_4,X_1\rangle$

Handling Lock Upgrade $X_1(A)$

A	$\{S_1\}$	$\langle\rangle$

$\longrightarrow$

A	$\{X_1\}$	$\langle\rangle$

A	$\{S_1,S_2,\ldots\}$	$\langle\rangle$

$\longrightarrow$

A	$\{S_1,S_2,\ldots\}$	$\langle X_1\rangle$

A	$\{S_1,S_2,\ldots\}$	$\langle X_4\rangle$

??? $\longrightarrow$

A	$\{S_1,S_2,\ldots\}$	$\langle X_4,X_1\rangle$

$\longrightarrow$

A	$\{S_1,S_2,\ldots\}$	$\langle X_1,X_4\rangle$

Priority to Lock Upgrades prevents a deadlock

T_4 cannot continue before T_1 issues $U_1(A)$. However, T_1 is itself suspended.

Handling $U_1(A)$

- Remove any lock held by T_1 on A.
- Grant outstanding lock requests if possible.

$$\boxed{A \mid \{X_1\} \mid \langle S_2,S_3,X_4,S_5\rangle} \longrightarrow \boxed{A \mid \{S_2,S_3\} \mid \langle X_4,S_5\rangle}$$

Effect of Lock Scheduling

- Consider again $W_1(A)R_2(A)R_3(B)W_1(B)$.
- With locks/unlocks added as required by 2PL, the execution order up-front $R_3(B)$ may be
  $X_1(A)W_1(A)X_1(B)U_1(A)S_2(A)R_2(A)U_2(A)$.
- This results in the lock table entry:

B	$\{X_1\}$	$\langle\rangle$

- When T_3 now asks $S_3(B)$, which is required in front of $R_3(B)$, T_3 is suspended:

B	$\{X_1\}$	$\langle S_3\rangle$

- T_1 continues with $W_1(B)U_1(B)$:

B	$\{S_3\}$	$\langle\rangle$

- T_3 ends with $R_3(B)U_3(B)$.
- Note that has $W_1(B)$ has been executed prior to $R_3(B)$.

Two-Phase Locking (2PL)

Multiple-granularity locking

Multiple-Granularity Locking

WEALTH

NAME	SUM
An	2
Bob	1
…	…

T_1: SELECT * FROM WEALTH

T_2: SELECT * FROM WEALTH WHERE NAME='Bob'

T_3: UPDATE WEALTH SET SUM=SUM+1 WHERE NAME='An'

- T_1 needs an S-lock on Wealth, T_2 an S-lock on Bob's tuple, and T_3 an X-lock on An's tuple.
- If tuples are the only unit of locking, then T_1 needs to lock each individual tuple, causing much overhead.
- On the other hand, if relations are the only unit of locking, then T_3 requires an X-lock on Wealth, prohibiting concurrent access.
- Solution: allow locks at both the relation and tuple level.

Warning Locks

- S_1(Wealth) can be accepted only if no tuple of Wealth is X-locked by any other trx.
- How can we efficiently decide whether no tuple of a relation is X-locked?
- The idea is to require that no trx can hold an X-lock on a tuple unless it holds an IX-lock (intension exclusive) on the relation that contains the tuple.
- Intuitively, the IX-lock on the relation 'warns' about the existence of an X-lock on a tuple.
- S_1(Wealth) is incompatble with an IX-lock on Wealth.

Warning Protocol (1)

- You may not X-lock a tuple without holding an IX-lock on the relation containing that tuple.
- You may not S-lock a tuple without holding an IS-lock or an IX-lock on the relation containing that tuple.
- Operations: IX_i(Relation) and IS_i(Relation).
- Used with 2PL in order to ensure serializability.
- E.g., W=Wealth, A=An's tuple, B=Bob's tuple:
  - T_1=S_1(W) ⟨read tuples of W⟩ U_1(W)
  - T_2=IS_2(W)S_2(B) R_2(B) U_2(B)U_2(W)
  - T_3=IX_3(W)X_3(A) W_3(A) U_3(A)U_3(W)

Warning Protocol (2)

- Compatibility matrix at relation level:

held \ asked	IS	IX	S	X
IS	Yes	Yes	Yes	No
IX	Yes	Yes	No	No
S	Yes	No	Yes	No
X	No	No	No	No

- The foregoing can be easily extended to hierarchies with more than two levels.

Phantom Problem

S(WEALTH);

SELECT *

FROM WEALTH;

INSERT INTO WEALTH VALUES('Ed', 5);

SELECT *

FROM WEALTH;

U(WEALTH);

- The right-hand trx inserts a so-called 'phantom record.'
  - It seems that it needs no locks.

- **Phantom problem:** the second read of the same relation gets more tuples.
- The schedule is definitely not equivalent to a serial one.
- Solution: you are not allowed to **insert a tuple** in a relation without holding an X-lock on the relation.

Two-Phase Locking (2PL)

Deadlock

Deadlock

- Concurrent execution of $T_1 = S_1(A)R_1(A)X_1(B)W_1(B)U_1(A)U_1(B)$ and $T_2 = S_2(B)R_2(B)X_2(A)W_2(A)U_2(B)U_2(A)$ can start as $S_1(A)R_1(A)S_2(B)R_2(B)X_2(A)$ resulting in:

A	$\{S_1\}$	$\langle X_2 \rangle$
B	$\{S_2\}$	$\langle\rangle$

- T_2 is suspended and T_1 continues with $X_1(B)$:

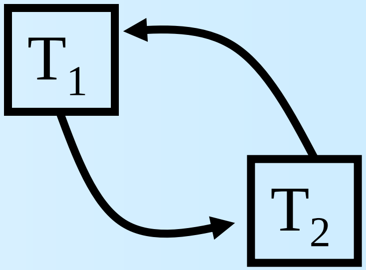
A	$\{S_1\}$	$\langle X_2 \rangle$
B	$\{S_2\}$	$\langle X_1 \rangle$

- Both T_1 and T_2 are suspended in a so-called deadlock.

Wait-For Graph

- Add a node labeled T_i for every trx T_i that holds a lock or is waiting for one.
- We say that T_i waits for T_k if T_i waits for a lock held by T_k or T_i follows behind T_k in some wait queue.
- Add an edge from node T_i to node T_k if T_i waits for T_k .
- There is a deadlock iff the wait-for graph is cyclic.
- For example,

A	$\{S_1\}$	$\langle X_2 \rangle$
B	$\{S_2\}$	$\langle X_1 \rangle$



Four Ways to Resolve Deadlocks

1. By timeout: Put a limit on how long a trx may be active, and if a trx exceeds this time, roll it back.
2. Maintain the wait-for graph at all times, and roll back any trx that makes a request that would cause a cycle.
3. Compute the wait-for graph periodically, and break cycles (if any) by rolling back trx.
4. Deadlock prevention by timestamps (cf. next).

Deadlock Prevention by Timestamps

- We associate with each trx a timestamp.
- We say that T is older than U (and U is younger than T) if the timestamp of T is smaller than U's timestamp.
- Wait-Die Scheme. If a younger trx makes a request that would cause it to wait for an older trx, then the younger trx is rolled back.
- Wound-Wait Scheme. If an older trx makes a request that would cause it to wait for a younger trx, then the younger trx is rolled back.

Wait-Die Example

- Assume that T_2 is younger than T_1.
  In general, assume that the timestamp of T_i is i.
- Concurrent execution of
  $T_1=S_1(A)R_1(A)X_1(B)W_1(B)U_1(A)U_1(B)$ and
  $T_2=S_2(B)R_2(B)X_2(A)W_2(A)U_2(B)U_2(A)$ can start as
  $S_1(A)R_1(A)S_2(B)R_2(B)X_1(B)$ resulting in:

A	$\{S_1\}$	$\langle\rangle$
B	$\{S_2\}$	$\langle X_1\rangle$

- T_1 is suspended and T_2 continues with $X_2(A)$.

A	$\{S_1\}$	$\langle X_2\rangle$
B	$\{S_2\}$	$\langle X_1\rangle$

? $\xrightarrow{\text{roll back } T_2}$

A	$\{S_1\}$	$\langle\rangle$
B	$\{X_1\}$	$\langle\rangle$

Wound-Wait Example

- Assume that T_1 is older than T_2.
- Concurrent execution of $T_1 = S_1(A)R_1(A)X_1(B)W_1(B)U_1(A)U_1(B)$ and $T_2 = S_2(B)R_2(B)X_2(A)W_2(A)U_2(B)U_2(A)$ can start as $S_1(A)R_1(A)S_2(B)R_2(B)X_2(A)$ resulting in:

A	$\{S_1\}$	$\langle X_2 \rangle$
B	$\{S_2\}$	$\langle\rangle$

- T_2 is suspended and T_1 continues with $X_1(B)$.

A	$\{S_1\}$	$\langle X_2 \rangle$
B	$\{S_2\}$	$\langle X_1 \rangle$

? $\xrightarrow{\text{roll back } T_2}$

A	$\{S_1\}$	$\langle\rangle$
B	$\{X_1\}$	$\langle\rangle$

Why Wait-Die Works

- In Wait-Die, trx can only wait for younger trx.
- Suppose the wait-for graph contains a cycle.
- One of the trx involved in the cycle is the youngest, say T.
- In the cycle, there must be an edge from T to some other trx, say U.
- But then U is younger than T, a contradiction.
- We conclude by contradiction that no cycle can exist.

Why Wound-Wait Works

- In Wound-Wait, trx can only wait for older trx.
- Suppose the wait-for graph contains a cycle.
- One of the trx involved in the cycle is the oldest, say T.
- In the cycle, there must be an edge from T to some other trx, say U.
- But then U is older than T, a contradiction.
- We conclude by contradiction that no cycle can exist.

No Starvation

- In both Wait-Die and Wound-Wait, it is always the younger trx that is rolled back.
- Trx that are rolled back, restart with their old timestamp, so that every trx is guaranteed to eventually complete.
- Note incidentally that Wait-Die never rolls back a trx that has acquired all the locks it needs.

Two-Phase Locking (2PL)

Strict 2PL

Dirty-Read Problem

- *Dirty-Data:* Data is called dirty if it has been written by a trx that is not yet committed.
- *Dirty-Read:* A read by trx T is dirty if it reads dirty data written by another trx.
- Dirty-Read problem:
  1. U writes a new value for Y
  2. T reads U's value for Y (a dirty read)
  3. T finishes and commits
  4. U is aborted (e.g., for deadlock reasons)
- Since the effect of T is based on a value of Y that never really existed, the overall effect will generally not be equivalent to any serial schedule.

Strict Locking

- The solution of the dirty-read problem: Conceal dirty-data from other trx.
- *Rule L4:* A trx must not release any X-locks until the trx has committed or aborted.

  Strict 2PL = 2PL + rule *L4*

Strict 2PL

- Strict 2PL is not deadlock free.
- Locking and unlocking can be transparant to programmers:
  1. the locking scheduler can insert lock actions into the stream of reads and writes;
  2. the scheduler releases locks only after the trx is committed or aborted.

TOC Concurrency Control

- Serializable schedules
- Two-Phase Locking (2PL)
- Concurrency control by timestamps
  - Basic idea
  - Thomas Write Rule
- Isolation levels in SQL2
- Exercises

Concurrency control by timestamps

Basic idea

Concurrency Control by Timestamps

- Assign to each trx T a unique timestamp, denoted TS(T), indicating the start time of T.
- Not the same timestamp as the one used for deadlock prevention.
- Timestamp-based scheduling will limit schedules to those that can be serialized into the serial schedule in which trx appear in ascending TS order.
- That is, if $TS(T_1) < TS(T_2) < \ldots < TS(T_n)$, then the schedule can be serialized into the serial schedule $[T_1, T_2, \ldots, T_n]$.

Read and Write Time

- Associate two timestamps with each database element Y:
  - o RT(Y), the read time of Y, which is the highest timestamp of a trx that has read Y.
  - o WT(Y), the write time of Y, which is the highest timestamp of a trx that has written Y.
- The idea is to abort trx issuing reads or writes that would result in a schedule that cannot be serialized into a serial schedule where trx appear in TS order.

Handling Read Requests

Suppose trx T issues $R_T(Y)$. Two cases can occur:

- $TS(T) < WT(Y)$. That is, some trx (say U) with $TS(T)<TS(U)$ has already written Y (and set the value of $WT(Y)$).

  We cannot accept the read, or else the schedule produced would be …$W_U(Y)$…$R_T(Y)$ which cannot be serialized into our intended serial schedule where T precedes U.

  Intuitively, the read comes too late…
- $TS(T) \geq WT(Y)$ causes no problem.

Handling Write Requests

Suppose T issues $W_T(Y)$.

- $TS(T) < RT(Y)$, i.e., some trx (say U) with $TS(T)<TS(U)$ has already read Y (and set the value of RT(Y)).

  We cannot accept the write, or else the schedule produced would be $\ldots R_U(Y)\ldots W_T(Y)$ which cannot be serialized into our intended serial schedule where T precedes U.
- $TS(T) < WT(Y)$, i.e., some trx (say U) with $TS(T)<TS(U)$ has already written Y (and set the value of WT(Y)).

  We cannot accept the write, or else the schedule produced would be $\ldots W_U(Y)\ldots W_T(Y)$ which cannot be serialized into our intended serial schedule where T precedes U.
- $TS(T) \geq RT(Y)$ AND $TS(T) \geq WT(Y)$ causes no problem.

Overview

- Trx T wants to read Y:

  if $\quad$ $TS(T) \geq WT(Y)$
  then $\quad$ execute the read
  $\quad$ $RT(Y) := \max(RT(Y), TS(T))$
  else $\quad$ abort T;

- Trx T want to write Y:

  if $\quad$ $TS(T) \geq WT(Y)$ AND $TS(T) \geq RT(Y)$
  then $\quad$ execute the write
  $\quad$ $WT(Y) := TS(T)$
  else $\quad$ abort T;

Concurrency Control by Timestamps Example

T_1	T_2	T_3	A	B	C
20	15	17	RT=0	RT=0	RT=0
			WT=0	WT=0	WT=0
$R_1(B)$				RT=20	
	$R_2(A)$		RT=15		
		$R_3(C)$			RT=17
$W_1(B)$				WT=20	
$W_1(A)$			WT=20		
	$W_2(C)$				
	Abort				
		$W_3(A)$			
		Abort			

Concurrency control by timestamps

Thomas Write Rule

Thomas Write Rule (1)

- Suppose T wants to write Y but TS(T) < WT(Y), i.e., some trx (say U) with TS(T)<TS(U) has already written Y. Accepting the write would produce $\ldots W_U(Y) \ldots W_T(Y)$ which cannot be serialized into a serial schedule where T precedes U.
- Can't we -- instead of aborting T -- simply <u>skip</u> $W_T(Y)$, pretending (i) that $W_T(Y)$ occurred ahead of $W_U(Y)$ in the right order, and (ii) that T's value for Y was overwritten by U later on?

pretend?

$W_U(Y) \ldots W_T(Y)$

Thomas Write Rule (2)

- Pretense is possible <u>unless</u> a trx V that should have read T's value for Y got another value instead. In fact, suppose TS(T)<TS(V)<TS(U) and

Pretend?

$R_V(Y) \ldots W_U(Y) \ldots W_T(Y)$

- We cannot simply pretend that $W_T(Y)$ occurred before $R_V(Y)$, because V did see another value for Y!
- Then, since V has read Y, the read time of Y must be at least TS(V). From TS(V)≤RT(Y) and TS(T)<TS(V), it follows TS(T)<RT(Y).
- If we require TS(T)≥RT(Y), then there can be no such V and we can safely pretend that the write of T occurred in order.

  Intuitively, TS(T)≥RT(Y) expresses that no read has 'missed' the value of the write that comes too late.

Overview With Thomas Write Rule

- Trx T want to write Y:

if $TS(T) \geq RT(Y)$

then
$$\begin{cases} \text{if} & TS(T) \geq WT(Y) \\ \text{then} & \text{execute the write} \\ & WT(Y) := TS(T) \\ \text{else} & \text{ignore the write (Thomas)} \end{cases}$$

else abort T;

Thomas Write Rule Example

T_1	T_2	T_3	A	B	C
20	15	17	RT=0	RT=0	RT=0
			WT=0	WT=0	WT=0
$R_1(B)$				RT=20	
	$R_2(A)$		RT=15		
		$R_3(C)$			RT=17
$W_1(B)$				WT=20	
$W_1(A)$			WT=20		
	$W_2(C)$				
	Abort				
		$W_3(A)$			

Ignored

Preventing Dirty-Reads

- The above timestamp-based scheduling decisions need to be extended by a mechanism to solve the dirty-read problem, i.e., to prevent a trx from reading data written by a concurrent uncommitted trx.
- The solution consists in suspending a trx that wants to read a dirty database element until the trx that has written the element has committed or aborted.

Restart

- Aborted trx may be restarted later on.
- If they restart with the same timestamp, then they will be aborted again.
- So aborted trx need to get a new timestamp when they are restarted.
- This is unlike the timestamps used in Wait-Die or Wound-Wait.

TOC Concurrency Control

- Serializable schedules
- Two-Phase Locking (2PL)
- Concurrency control by timestamps
- Isolation levels in SQL2
- Exercises

Transaction Support in SQL2

- The SQL2 standard does not assume that every trx runs in a serializable manner.
- The user can set an isolation level for each trx.
- Isolation levels are characterized in terms of Dirty-Read, Non-repeatable Read, and Phantom Read.
- Recall *Dirty-Read:*

1. T_1 modifies db element Y;
2. T_2 reads Y before T_2 is commited or aborted;
3. If T_2 is rolled back, T_1 has read a value for Y that was never committed and so never really existed.

Non-repeatable Read

- *Non-repeatable Read:*
  1. T_1 reads a db element Y.
  2. T_2 writes a new value for Y, or deletes Y, and commits.
  3. T_1 reads Y again and discovers that it has been modified or deleted.

  This series of events is non-serializable and impossible in 2PL -- assuming that you cannot delete a db element without holding an X-lock on it.

Phantom Read

- Recall *Phantom Read:*
  1. T_1 reads a set of database elements specified by a SELECT-query.
  2. T_2 inserts new db elements and commits.
  3. T_1 gets a different result for the same query.
- Note that Phantom generalizes Non-repeatable Read to **sets** of db elements.
- Phantom Reads may occur in a system that prevents Non-repeatable Reads.

Isolation Levels in SQL

- Four isolation levels can be set by the **SET TRANSACTION** command.

Level \ Phenomenon	Dirty Read	Non-repeatable Read	Phantom Read
READ UNCOMMITED	possible	possible	possible
READ COMMITTED	impossible	possible	possible
REPEATABLE READ	impossible	impossible	possible
SERIALIZABLE	impossible	impossible	impossible

TOC Concurrency Control

- Serializable schedules
- Two-Phase Locking (2PL)
- Concurrency control by timestamps
- Isolation levels in SQL2
- Exercises

Exercise 1

- Given the schedule $S = R_1(C)R_1(A)W_2(B)R_2(A)W_1(D)W_2(C)W_1(A)$, determine whether 2PL allows the reads and writes to occur **in exactly the order shown**.

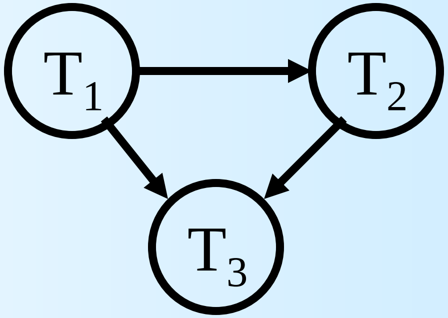
  Answer:

- Since $R_1(C)$ precedes $W_2(C)$, the precedence graph contains an edge from T_1 to T_2.
- Since $R_2(A)$ precedes $W_1(A)$, the precedence graph contains an edge from T_2 to T_1.
- Since the precedence graph contains a cycle, the schedule is not serializable, and hence impossible in 2PL.

Exercise 2

- Given the schedule
  $S=W_1(A)R_2(A)W_1(B)W_3(A)W_2(B)$,
  determine whether 2PL allows the reads and writes to occur **in exactly the order shown**.

  Answer:
- The precedence graph is:



- The schedule is serializable. But recall that not all serializable schedules are 2PL-schedules.

Exercise 2 (Cntd.)

$X_1(A)W_1(A)X_1(B)U_1(A)S_2(A)R_2(A)W_1(B)U_1(B)$
$X_2(B)\ U_2(A)X_3(A)\ W_3(A)U_3(A)W_2(B)U_2(B)$

- It is easy to see that T_1, T_2, and T_3 each satisfy rules *L1* and *L2*:
  1. $T_1 = X_1(A)W_1(A)X_1(B)U_1(A)W_1(B)U_1(B)$
  2. $T_2 = S_2(A)R_2(A)X_2(B)\ U_2(A)W_2(B)U_2(B)$
  3. $T_3 = X_3(A)\ W_3(A)U_3(A)$
- As for rule *L3*, T_1 issues $U_1(A)$ prior to $S_2(A)$, and T_2 issues $U_2(A)$ prior to $X_3(A)$.

  Also, T_1 issues $U_1(B)$ prior to $X_2(B)$.

Exercise 3

- Assume the following lock table:

db element	locks held	wait queue
A	$\{S_1,S_2\}$	$\langle X_1,X_3\rangle$
B	$\{X_1\}$	$\langle S_2\rangle$

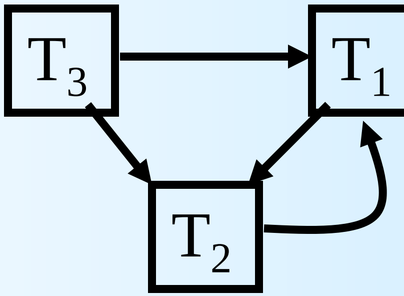
- Which actions in a system ensuring 2PL could have resulted in this lock table?
- Since all three trx are suspended, a deadlock has occurred. Which trx need to be rolled back?
- Explain how this deadlock would have been prevented (i) by Wait-Die, (ii) by Wound-Wait.

Exercise 3 (Cntd.)

$S_1(A)\ S_2(A)\ X_1(B)\ S_2(B)\ X_1(A)\ X_3(A)$

db element	locks held	wait queue
A	$\{S_1, S_2\}$	$\langle X_1, X_3 \rangle$
B	$\{X_1\}$	$\langle S_2 \rangle$

Wait-for graph:



Either T_1 or T_2 must be rolled back.

Note that T_3 is suspended but is not part of a cycle.

Exercise 3 (Cntd.)

- Same sequence with Wait-Die.

$S_1(A)$ $S_2(A)$ $X_1(B)$ $S_2(B)$

db element	locks held	wait queue
A	{S_1, $\cancel{S_2}$}	⟨ ⟩
B	{X_1 }	⟨S_2⟩

T_2 is rolled back…

Exercise 3 (Cntd.)

- Same sequence with Wound-Wait.

$S_1(A)\ S_2(A)\ X_1(B)\ S_2(B)\ X_1(A)$

db element	locks held	wait queue
A	$\{ X_1, \cancel{S_2} \}$	$\langle X_1 \rangle$
B	$\{ X_1 \}$	$\langle \cancel{S_2} \rangle$

T_2 is rolled back…

Transaction Management
END

…