Today:

- Other lock schemes: increment locks.
- View-serializability: transactions can be serializable even if they fail the conflict-serializable test.
- Timestamps as a concurrency mechanism.

Soon:

- Validation as another concurrency mechanism.
- Logical logging (explain what happened at an abstract level).
- Deadlocks.
- Distributed DB's:
  - Distributed locking.
  - Distributed commit.

Increment Actions and Locks

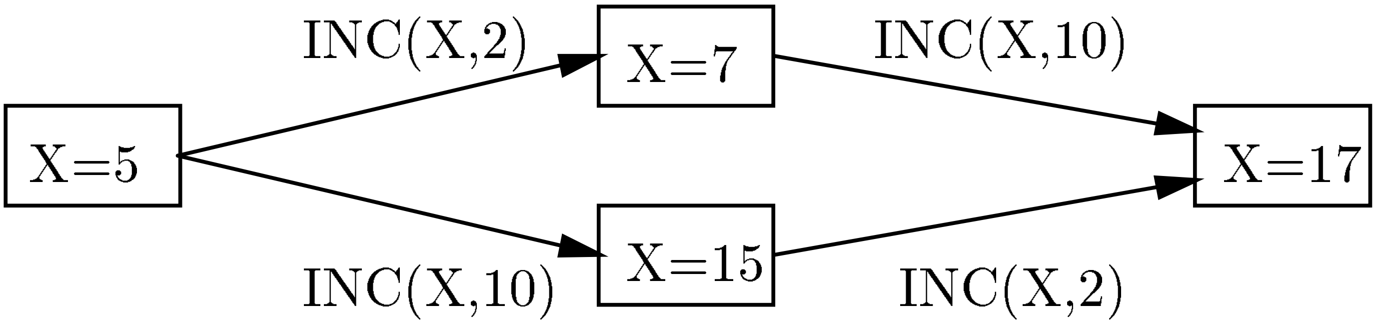
Some applications depend heavily on an *increment* operation, which is an atomic execution of `READ(X,t); t := t+c; WRITE(X,t);` for some constant c.

- Example: A banking system will frequently increment accounts (including deductions, which are increments with $c < 0$).
- As an action: `INC(X,c)`.
- Shorthand notation: $inc_i(X, c)$ = "transaction T_i increments DB element X by constant c."
- As a lock: $il_i(X)$.
    - ❖ Consistency rule: `INC` action permitted by T_i only when T_i holds an il lock.

Conflicts Involving `INC` Actions

`INC(X,c)` does not commute with `READ(X)`, `WRITE(X)`.

- But it does commute with `INC(X,d)`!



Compatability Matrix for Shared, Exclusive, Increment Locks

	S	X	I
S	yes	no	no
X	no	no	no
I	no	no	yes

Problem

Since S and I locks are each compatible with only themselves, why can't we just merge those two lock modes?

View Serializability

- Conflict-serializability is "conservative"; if it says a schedule is serializable, then it surely is, no matter what the transactions actually do with their data.
- The ultimate serializability test would take into account what the transactions actually do.
  - Example: remember how we were able to interleave two transactions in seemingly illegal ways because one transaction just mutliplied A and B by 1.
- *View serializability*: a middle ground that focuses on how values are transmitted from writers to readers.
  - Exploits the possibility that a value written may be overwritten without having had any effect on any transaction.

View Serializability Rules

1. If T_i is the last to write X before T_j reads X, then T_i must precede T_j.

2. If T_i writes X, and that value of X is read by T_j, then any T_k that also writes X must come *either* before T_i or after T_j.

3. The above rules hold also if we introduce a fictitious transaction T_0 that at the beginning of the schedule writes all DB elements and T_f that reads them all at the end.

   - These represent the DB state before and after the scheduled transactions.

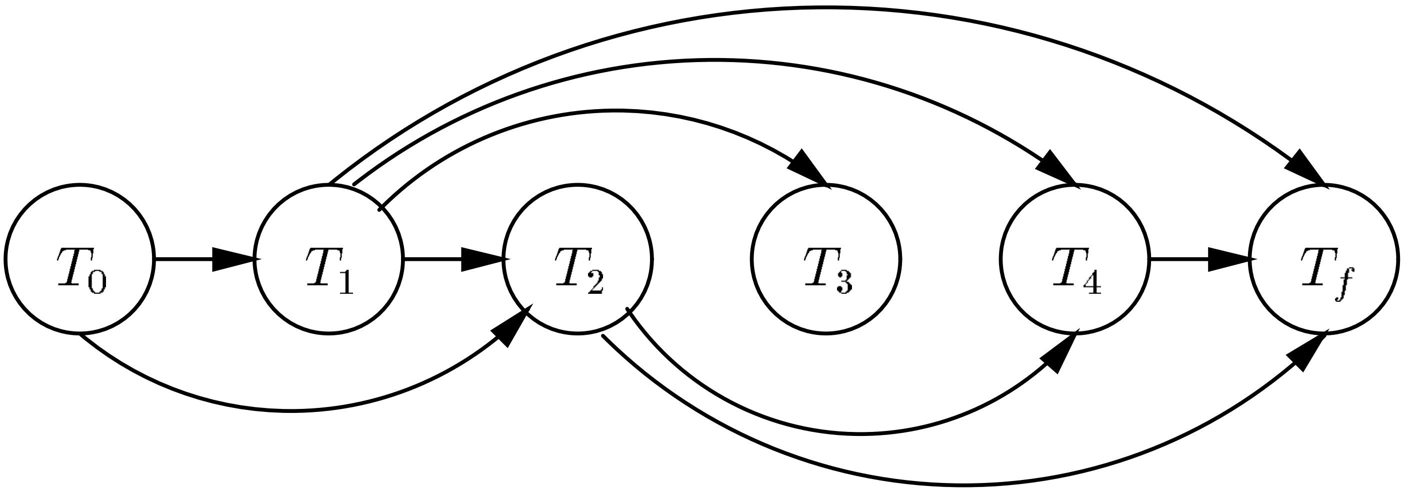
The Polygraph

- Nodes = transactions + fictitious T_0 and T_f.
- Arc $T_i \rightarrow T_j$ if T_i writes a value T_j reads.
- Arc pair (dotted) from $T_k \rightarrow T_i$ and $T_j \rightarrow T_k$ if T_j reads an element written by T_i, and T_k also writes that element.
  - ❖ But if T_i is T_0 or T_j is T_f, then one of these arcs is impossible, so just put the alternative arc as a single (not dotted) arc.

Example

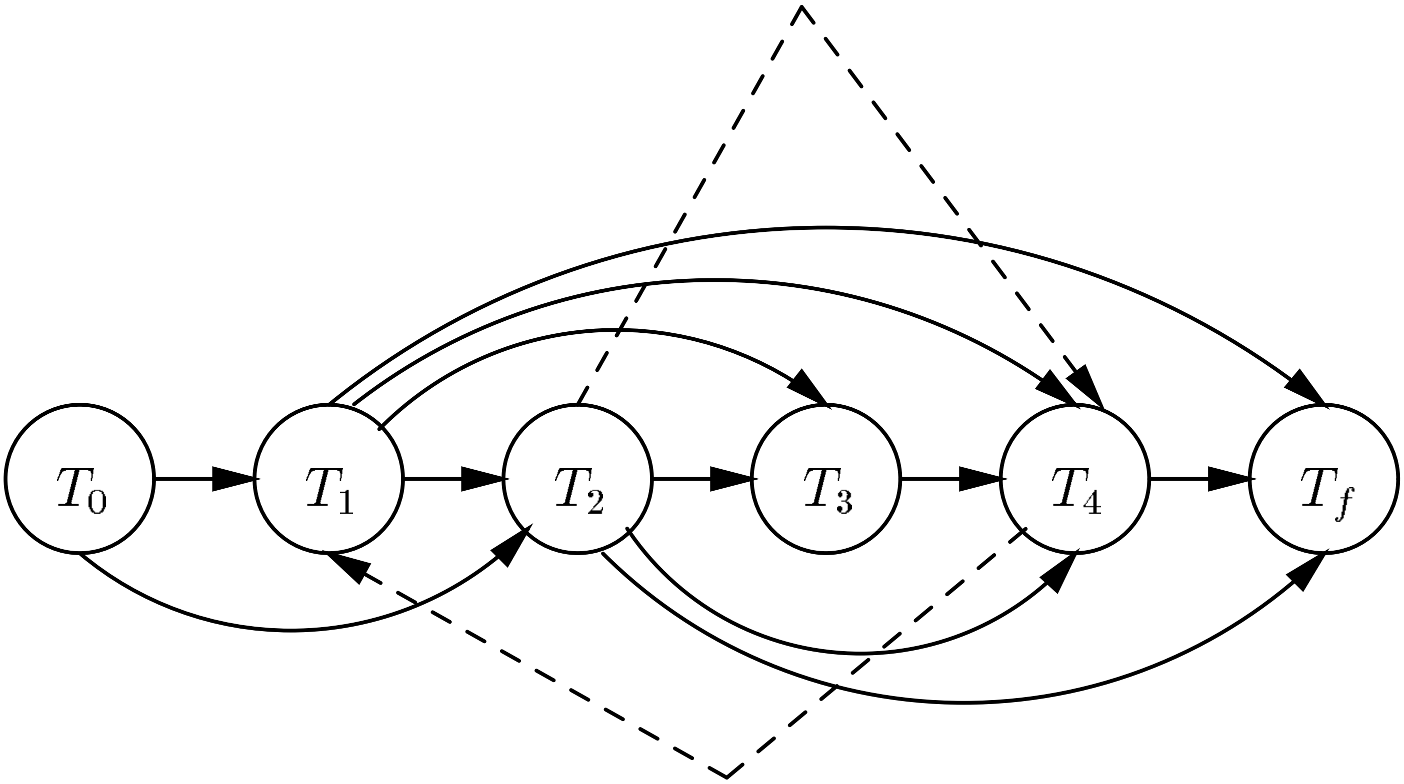
T_1 :		r_1A	w_1C		w_1B							
T_2 :	r_2A								w_2D	r_2B		
T_3 :				r_3C			w_3A					
T_4 :						r_4B		r_4C			w_4A	w_4B

- The arcs due to read-write relationships:



Potential conflicts

- A is written by T_0, T_3, and T_4.
  - The value written by T_3 is *never* read, so there is no constraint on where T_4 could go in a serial schedule.
  - But the value written by T_4 is read by T_f. Since T_3 cannot appear after T_f, it must appear before T_4.
  - The value written by T_0 is read by T_1 and T_2. Since T_3 and T_4 cannot appear before T_0, they must appear after T_1 and T_2.
- B is written by T_0, T_1, and T_4.
  - The value written by T_0 is not read.
  - The value written by T_4 is read only by T_f, so we need an arc $T_1 \to T_4$.
  - The value written by T_1 is read by T_2 and T_4. Thus, T_4 cannot appear between T_1 and T_2, resulting in the arc pair $(T_4 \to T_1,\ T_2 \to T_4)$.


T_0
T_1
T_2
T_3
T_4
T_f

Test for View Serializability

- A polygraph is *acyclic* if there is some choice of one arc from each arc-pair that turns the polygraph into an acyclic (ordinary) graph.
- A schedule is view-serializable if its polygraph is acyclic.

Example

For our example polygraph, picking $T_4 \rightarrow T_1$ is wrong; it produces a cycle. But $T_2 \rightarrow T_4$ works.

- The resulting graph is acyclic and has the topological order $T_0, T_1, T_2, T_3, T_4, T_f$.

Timestamps

- Unique value representing a time.
  - Example: clock time.
  - Example: serial counter.
  - Even distributed systems can have timestamps, e.g., high-order bits = clock time of local machine; low-order bits = ID for that machine.

Serializability Via Timestamps

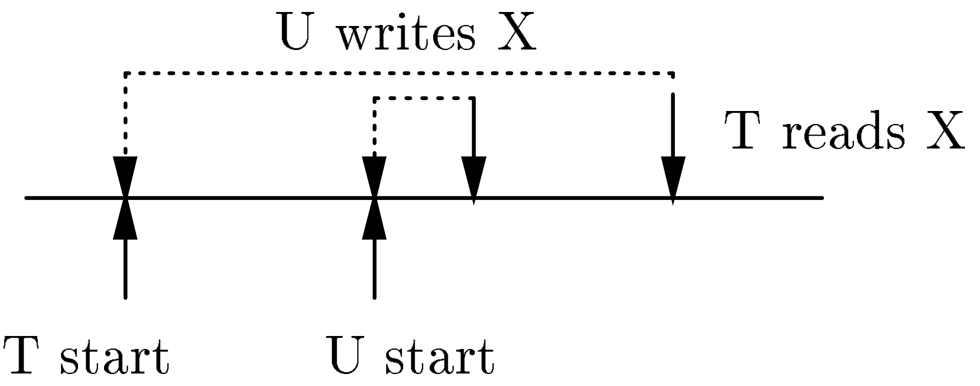
Big idea: let things rumble along without any locking or scheduling. As transactions read/write, check that what they are doing makes sense if the serial order was the same as the order in which transactions initiated.

- ❖ Big win if most transactions are read-only.

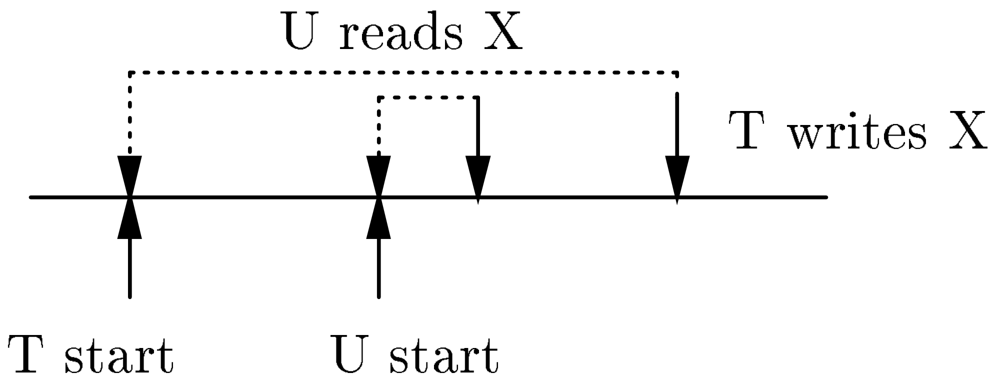
- Every transaction T is given a timestamp $TS(T)$ when it initiates.
- Every DB element X has two timestamps:
  1. $RT(X)$ = highest timestamp of a transaction to read X.
  2. $WT(X)$ = highest timestamp of a transaction to write X to the DB.
- Every DB element X has a bit $C(X)$ indicating whether the most recent writer of X has committed.
  - ❖ Essential to avoid a "dirty read," where a transaction reads data that was written by a transaction that later aborts.

What is Physically Unrealizable?

- Transaction T tries to read X, but $TS(T) < WT(X)$.
    - ❖ T would read something that was written after T ostensibly finished.

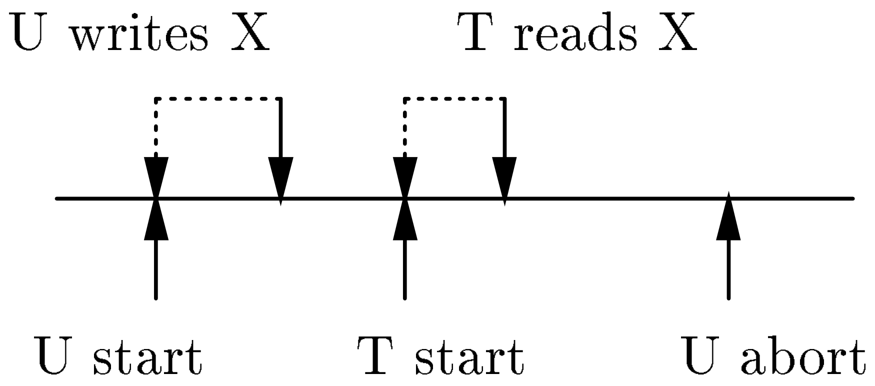


- Transaction T tries to write X, but $RT(X) > TS(T) > WT(X)$.
  - ❖ Some other transaction read a value written earlier than T, when it should have read what was written T.



But Wait if Data Might be Dirty

- T tries to read X, and $TS(T) > WT(X)$, but $C(X) =$ false.
    - ❖ T would be reading dirty data — a risk we won't take.



- Another approach: don't use $C(X)$ bit, but be prepared to perform *cascading rollback* using an undo log.
    - ❖ If transaction T aborts, examine the log for any transaction that used data written by T and (recursively) roll that back too.

Abort/Update Decision

- If the read is legal, perform it and change $RT(X)$ if $TS(T)$ is greater than the old value.
- If a write is legal, do nothing if $WT(X) > TS(T)$. If $WT(X) < TS(T)$, perform the write and change $WT(X)$ to be $TS(T)$.
- If illegal, *rollback* T = abort T and restart it with a new timestamp.

Commitment

- When a transaction finishes with no rollback, *commit* the transaction by changing all $C(X)$ bits to true.

Example

T_1	T_2	T_3	A	B	C
200	150	175	$RT = 0$ $WT = 0$	$RT = 0$ $WT = 0$	$RT = 0$ $WT = 0$
$r_1(B)$;				$RT = 200$	
	$r_2(A)$;		$RT = 150$		
		$r_3(C)$;			$RT = 175$
$w_1(B)$;				$WT = 200$	
$w_1(A)$;			$WT = 200$		
	$w_2(C)$;				
	$abort$;				
		$w_3(A)$;			

- T_2 aborts because it tries to write "at time 150" when another value of C was already read "at time 175."
- T_3 is allowed to "write A," but since there is already a later write of A, the DB is not affected.

Timestamps Vs. Locks

- Locking requires a lock table for currently locked items, while timestamping uses space for two timestamps in each DB element, locked or not.
- Locking may cause transactions to wait; timestamping doesn't cause waiting, but may abort transactions.
  - Net effect: if most transactions are read-only or few transactions interfere, then timestamping gives better throughput; otherwise not.